Tolerant (Parallel) Programming with F-Nets and Software Cabling

David C. DiNucci
MRJ Technology Solutions, Inc.
NASA Ames Research Center, M/S T27A-2
Moffett Field, CA 94035

Abstract

In order to be truly portable, a program must be tolerant of a wide range of development and execution environments, and a parallel program is just one which must be tolerant of a very wide range. First, the term "tolerant programming" is defined. Then, a formal model called F-Nets is described in which parallel algorithms are expressed as folded partial-orderings of operations, and this is argued to provide a suitable framework for building tolerant programs. Finally, Software Cabling (SC), a very-high-level graphical programming language, demonstrates how many of the features normally expected from today's computer languages (e.g. data abstraction and data parallelism) can be obtained within the F-Net paradigm.

1. Introduction

The subgoals of parallel processing are very similar to the subgoals of software engineering in general—i.e. the decomposition of a large problem into smaller tasks or modules, the precise expression of the scope of data and the semantics of sharing and communicating data efficiently and safely between modules. In fact, differentiating parallel software engineering from traditional software engineering can be a mistake, since doing so may lead one to believe that a parallel program is engineered in a different (and perhaps even more roundabout) way than a "real" program. Parallel software engineering will only come into its own when parallel programs are considered "real", and the ability for a program to run efficiently on a parallel machine is just another desirable feature which the software possesses. A corollary is that tools, languages, and methodologies for "parallel" programming must be useful enough to facilitate any kind of programming.

This may seem like a lot to ask. After all, parallel software engineering seems to lag behind sequential in virtually every area: formal models, languages, tools, and development and debugging strategies. Of these, formal models constitute the linchpin. With an adequate formal model, the other methodologies can be developed to exploit the properties of that model. To facilitate the needs above, the model must be tolerant of, but not dependent upon, the sharing of memory or the passing of messages. Likewise, it must be adaptable to different languages, different architectures, different amounts of parallelism (including none at all), and the possible presence of nondeterminism.

The objective of this paper is to present a tolerant approach to software engineering. That is, rather than engineering a product for a particular parallel environment, the goal is to engineer one which is tolerant of many different environments, including parallel environments. The paper will first describe the full meaning of tolerant programming, then will present a simple yet formal model for (parallel) computation, called F-Nets, which embodies that meaning and thereby serves as the basis for other tools. Then, a very-high-level graphical programming language, called Software Cabling (SC), is presented as a case study to demonstrate that the F-Net approach can be used to approach real-world problems.

2. Tolerant Programming

Although the term "tolerant programming" could be used to describe tolerance to any number of traits, those covered here are concurrency, latency, semantics, changing environment, bugs, and language. Tolerance of these traits refers to the ability of a program or programming methodology to work well independently of the values of these traits. Some techniques which might be used to obtain tolerance are described below, but tolerance may be achieved through either user effort or automatically through tools (e.g. compilers). In fact, tolerance often relies on the ability to infuse the program with potentially useful information, and this may not be possible without the aid of tools.

2.1. Concurrency

Concurrency tolerance refers to providing the opportunity for scaling while not sacrificing efficiency on less-scalable platforms. For example, a large program can be decomposed into relatively small parts which may be able

to execute concurrently, but there should also be a mechanism to efficiently recompose these into larger portions when parallel execution is not possible.

To express program decomposition, constructs which are well defined and easily understood in both sequential and parallel environments should be used. Traditionally, loops have filled this role, but there are better examples, such as functions and atomic transactions.

2.2. Latency

Latency tolerance refers to the ability to adapt to high-latency environments without impacting efficiency in low-latency environments. Such tolerance is gained primarily by providing information about the expected data movement patterns. For example, whenever possible, the programmer should:

1) Express need for data long before it is used (and maybe even before process starts running—i.e. staging),

2) Express where newly created or modified data will be used next, even before it is requested there,

3) Refrain from waiting for last version of data to be consumed before creating next version (i.e. queueing),

4) Block (group) data to allow multiple data items to move as a unit whenever it is requested or forwarded, thus cutting latency/byte,

5) Pipeline computation.

2.3. Semantics

Semantic tolerance refers to the recognition that different environments may have different default semantics, so the desired semantics (or lack thereof) should be clearly expressed. For example, shared memory and message passing are different standard semantic combinations for communication, and using one combination globally throughout the program can make the program much less efficient in an environment which implements the other by default. A better approach is to delineate the application's semantics for each data communication, so that the required semantics can be implemented in the most efficient way in the available environment—e.g.

1) Destructive read (i.e. read on non-empty, dequeue)

2) Destructive write (i.e. standard over-write)

3) Non-destructive write (i.e. enqueue), or

4) Non-destructive read (i.e. standard read)

In some cases, it is appropriate to express the lack of a specific required semantics, especially as it relates to the ordering of operations. For example, a designer may not care about the order in which some commutative arithmetic operations are performed, nor perhaps the order in which some outputs are produced. Such an expression of acceptable non-determinism may make a program run faster in environments where loads or processor speeds vary. It is important, however, to ensure that non-determinism does not sneak into the design accidentally.

2.4. Changing Environment

Changing Environment tolerance refers to adaptation to an environment which changes as the program is running. Environment Tolerance includes the well-known Fault Tolerance, which is tolerance to processors which are made unavailable in an abrupt and unannounced fashion, but it also includes tolerance to any circumstances when processors leave in a pre-announced and well-defined way, and/or when processors become available which were not previously present. This tolerance takes that the computer(s) will likely be shared by many users, so the programming and/or scheduling method should allow for efficient sharing of limited resources.

2.5. Bugs

Bug tolerance relates to the provision of adequate debugging tools and mechanisms which provide the ability to find bugs and which limit the scope of bugs if they exist so that a small bug in one part of the program will have limited effects in other parts. This includes the ability to find unwanted nondeterminism and to efficiently record desired non-deterministic choices made during execution of a parallel program so that it can be reliably debugged and analyzed in a cyclic manner.

2.6. Language

Language tolerance describes the ability of a programming methodology to retain its utility even when the specific programming language used for implementation changes. This allows the implementor the flexibility to use the appropriate language for the job, or to change the language for any number of reasons, without requiring totally different engineering methodologies.

3. F-Nets

This section will informally describe a model for parallel computation called F-Nets. Although this model is not widely known, it is similar in many ways to Petri Nets[10], Turing Machines, Finite State Machines, dataflow, and CCS[9], all of which are well known. It was originally developed as a refinement of the Large-Grain Data Flow (LGDF) parallel programming approach[1], and early versions went by the name LGDF2 [6]. A full, formal description is available[5], but the informal English description which follows will be sufficient to illustrate the value of the

model.

The model will be described in six parts: its syntax, an operational semantics, an axiomatic semantics, its relationship to other models, its efficient expression on real computers, and some practical and theoretical properties it possesses which relate to tolerant programming.

3.1. Syntax

An F-Net can be considered as being similar to a Turing Machine, having a (possibly-infinite) tape, separated into *squares*. Each square contains a mutable value, called the *data state* of the square, and a mutable color, called its *control state*. Call the set of all possible data states *S*.

Along with the tape, the F-Net consists of a possibly-infinite set of *transitions*. Associated with each transition is a set of colored (non-white) *heads*. Each head is permanently attached to one square—i.e. the tape does not move relative to the heads. Each head is either a `read` head, a `write` head, a `read-write` head, or a `nodata` head (i.e. a head having neither read nor write capability). The heads for a given transition are enumerated from 1 to *n*, where *n* can be different for each transition. No two heads from the same transition are attached to the same square. In figure 1, the transitions are shown as circles, the heads as lines, and the colors as shadings (green shown as black). Arrowheads represent whether the head is `read` (at the transition end), `write` (at the tape end), `read-write` (at both ends), or `nodata` (neither end).

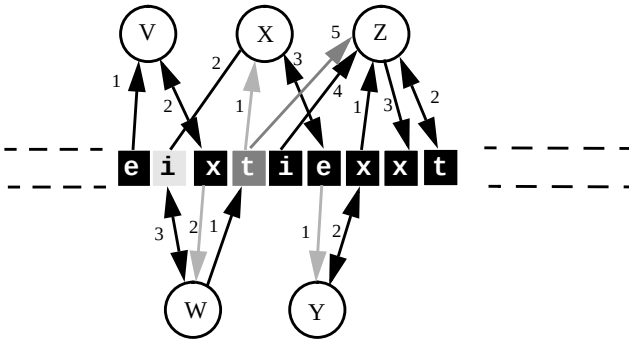


Figure 1. Sample F-Net in the Turing Machine style

Each transition has an associated *firing function*, which can be visualized as a table. If the number of `read` and `read-write` heads on the transition is *r*, then the table has $r^{|S|}$ entries, indexed by the possible combinations of *r* symbols under those heads. Each entry contains *n* colored (or white) symbols—i.e. one symbol corresponding to each head. Table 1 shows a sample firing function for transition W in the previous example, assuming S = {e,x,i,t}.

Any `read` or `nodata` head can be declared "predictable *c*" which is a declaration that the symbol corresponding to the head in all of the firing function entries has the color *c*. In the example, head 2 of W could be declared "predictable green".

Index		Contents		
2	3	1	2	3
e	e	e	x	e
e	x	e	x	x
e	i	e	x	i
e	t	e	x	t
x	e	x	x	x
x	x	x	x	i
x	i	x	x	t
x	t	x	x	e
i	e	i	x	i
i	x	i	x	t
i	i	i	x	e
i	t	i	x	x
t	e	t	x	t
t	x	t	x	e
t	i	t	x	x
t	t	t	x	i

Table 1. Sample Firing Function for Transition W

3.2. Operational Semantics

An F-Net works as follows. The machine begins in an initial state consisting of a predetermined symbol σ_q and the color green on each square *q*. Then, repeatedly, a *ready* transition is located (subject to the *liveness/fairness* rule below) and *evaluated* until there are no more ready transitions. A ready transition is defined as one for which each head is the same color as the square to which it is attached. Evaluation (or *firing*) consists of finding the entry in the table corresponding to the symbols under the `read` and `read-write` heads, and replacing the color of the squares under all heads with the color of the corresponding symbol from the table entry. For each write and read-write head, the symbol (i.e. data state) of the square is also changed to the corresponding symbol from the table entry.

The liveness/fairness rule states that if a transition is ready, then either it or some other ready transition which shares a square with it must be evaluated eventually (i.e. will not be eternally preempted) in the repeat cycle described above.

Note that for `read` and `nodata` heads, the symbols in the table serve no purpose other than as place-holders for the color at those positions, and that for predictable heads, even the colors in the table are superfluous. Note also that if any square becomes colored white during the course of an F-Net execution, no transitions attached to that square will ever be ready again, since heads cannot be white.

There is no need to consider the relative position of squares on the tape, so F-Nets are often represented graphically with the squares separated into disjoint rectangles. Although the firing function is rarely represented graphically, the colors which might be assigned to each head are often represented by colored dots near the connection between the line (head) and the tape square (rectangle).

`Read` and `nodata` heads with only one colored dot are implied to be predictable. In a black and white medium such as this, colors are sometimes represented by adding a "`/c`" suffix to the colored entity (i.e. dot, head, or within the firing function), where `c` represents the color (e.g. `g` for green, `b` for blue, `r` for red). See figure 2 for a more standard representation of the F-Net from figure 1.

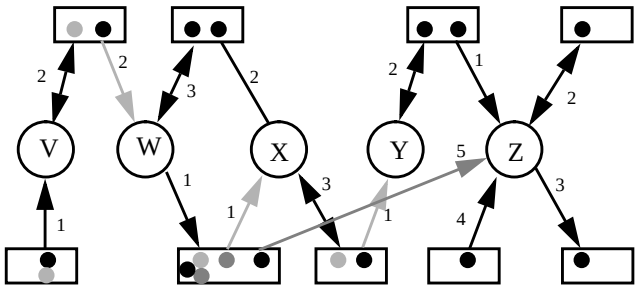


Figure 2. Same Sample F-Net in Standard Form

3.3. Axiomatic Semantics

Instead of describing an F-Net as an operational machine-like entity, it can be described as a means of parameterizing the set of computations which may be produced by the F-Net. The specification of the semantics then becomes a description of this parameterization—i.e. a characterization of the computations which can be produced by a given F-Net. Here, this characterization will take the form of a set of axioms.

An F-Net computation takes the form of a directed bipartite graph which is effectively a trace of the execution which would result from executing the F-Net using the operational semantics in the previous subsection and recording each transition firing, and the color and data on each tape square after each firing. It consists of state nodes (represented by squares) and firing nodes (represented by circles) connected by directed numbered edges (lines). Each state node corresponds to a tape square in the F-Net, and each firing node corresponds to a transition in the F-Net. Each state node is labeled with a control state (i.e. color) and a data state (from S). The axioms which constrain the computation follow:

Initial Conditions: Each tape square *q* in the F-Net will be represented by at least one state node in the computation graph from which any other state node for tape square *q* can be reached by following directed arcs. This node will have a control state labeling of green and a data state labeling of σq

Atomicity: Each state node in the computation graph will have at most one incoming edge and at most one outgoing edge.

Firing: If transition *t* in the F-Net has *n* heads, then for every firing node corresponding to transition *t*:

(a) There will be *n* incoming edges, numbered 1 to *n*, and *n* outgoing edges, numbered 1 to *n*, where each edge numbered *i* will be attached to a state node corresponding to the tape square under head *i* of transition *t*.

(b) The control state labeling of each state node attached to an incoming edge numbered *i* will be the same as the color of head *i* of transition *t*.

(c) If head *i* of transition *t* is a `nodata` or `read` head, then the state nodes attached to edges numbered *i* will have the same data state labels. The data states of the state nodes on all other edges and the control states of the outgoing edges will correspond to a single line in the firing function table of transition *t*, where the incoming edges correspond to the "index" columns and the outgoing edges corresponds to the "content" columns.

Liveness: If there is some transition *t* and some set of state nodes corresponding to the tape squares under the its heads such that the control state labeling of each state node corresponds to the color of the head of transition *t* attached to that square, then at least one of those state nodes must have an outgoing edge.

Time: The graph will be acyclic.

See figure 3 for an example of an execution graph for the F-Net in figure 2. By removing the state nodes from the execution graph (merging the two edges on each into a single edge), the execution represents a partial ordering of operations (where each operation is a transition firing function), or identically, a single function which is the composition of each of the involved firing functions.

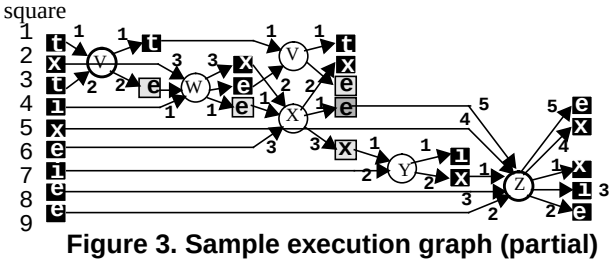


Figure 3. Sample execution graph (partial)

3.4. Relationship to Other Models

The description of the F-Net model above was based loosely on the Turing Machine. In fact, it is more properly interpreted as a TM coordination model—i.e. each transition can be considered a TM, and each F-Net square can be considered as a TM tape being passed from one TM to another. The next subsection will go into more detail on how this interpretation influences implementation on traditional computers.

A kinship between F-Nets and Petri Nets is suggested by the similar appearance (though the notations are confused). Specifically, an F-Net with only `nodata` heads (and therefore only predictable heads, since each firing function can

contain only a single table entry) can always be expressed as a Petri Net by mapping each F-Net transition to a Petri-Net transition and each F-Net square to *p* Petri-Net places, where *p* is the number of colors which the square might assume. Each head in the F-Net becomes two arcs in the Petri Net: an input arc from the place representing the color of the head, and an output arc to the place representing the predictable color which the head assigns to the square. The initial marking of the Petri Net is with one token on each "green" place. The ability to represent more complex F-Nets as Petri Nets depends upon the ability to represent each of the firing functions, which may not be useful, nor even be possible in the general case where some of the firing functions are partial recursive and/or S is infinite.

The presence of data in the F-Net model also makes it similar to dataflow or functional models. However, unlike these models which utilize simple arcs that are usually represented with single- (or zero-) assignment variables within languages, the squares in an F-Net represent updatable containers more like traditional imperative variables. Also, F-Nets more naturally accommodate non-determinism.

Each square in an F-Net can be considered as a finite state machine (with the state represented by the color) which also carries data, and these are interconnected with (partial) functions or Turing Machines. This suggests similarities and differences between F-Nets and Milner's CCS. Both can be considered as composing finite state machines, and upon atomic events which occur only when the states of different machines occur in pre-determined combinations. In both, these events in turn cause or allow each of the involved machines to perform a transition. However, in CCS, data transformation events are coupled with the machine transitions, whereas in F-Nets, they are coupled with the atomic events between the machines. It seems plausible that CCS theory could carry over to F-Nets.

By restricting F-Nets to use only green control states, a model similar to Unity[2] emerges. Guards in the latter are effectively replaced by control states in the former.

3.5. Real-World Implementation

To understand how the F-Nets model pertains to tolerant programming, it is first important to understand how its components manifest themselves in the real world. Only two such components will be discussed here: the firing functions and the tape squares. The remainder of the F-Net is typically represented in the same graphical form previously described.

In a standard computer, the firing function of each transition of an F-Net is normally implemented as a small deterministic (usually sequential) subprogram. That is, instead of being a table, the function is represented by the mapping of inputs to outputs implied by the code. The data state of each square is implemented as a standard data structure. This leaves only the colors (i.e. control state) to be handled in a somewhat non-standard manner.

When a transition fires, the data state of the squares under the transition's heads are passed to the subprogram as arguments. Although `nodata` heads can be neither read nor written, they are also supplied as a special kind of argument. At any time during the execution, the subprogram can execute a special statement, of roughly the form

`give` *`square color`*

which declares that the subprogram will make no further accesses to argument *`square`*, and that the square should be assigned the new color *`color`*. By executing one such statement for each of its arguments (i.e. squares), the subprogram expresses a mapping from the initial state of its read and read-write squares to the final state of its `read` and `read-write` squares and the new color for all of its squares, as it is required to do by the formal F-Net semantics. If a transition subprogram does not execute a `give` statement for some of its arguments, those squares effectively become white.

The word "repeatedly" in the semantics (third sentence) suggests that only one transition can be evaluating at a time, but this leads to both practical and theoretical problems. Practically, requiring sequential execution obviously decreases the model's value in the realm of parallel processing. Theoretically, sequential execution would require that a scheduler know when the evaluation of a transition subprogram was finished so that it could know that the next ready transition could be initiated. This means that, at every point in time, the scheduler would be required to decide whether further evaluation of the subprogram might lead to execution of more `give` statements (i.e. for arguments for which they had not already been executed). In other words, a correct sequential scheduler would be required to either solve the (impossible to solve) halting problem, or to conceivably let subprograms which never execute `give` statements for some arguments execute forever and thus contradict the required liveness properties.

Fortunately, the operational semantics in subsection B can be shown to be identical to these revised semantics:

> "An F-Net works as follows. The machine begins in an initial state ... . Then, repeatedly, a ready transition is located and *initiated*. A ready transition is .... . Initiation means changing the color of all the squares under the transition's heads to white, and then beginning evaluation of the transition. Evaluation ..."

In this case, the scheduler does not need to wait for one transition to finish its evaluation before initiating the next. A database theorem about two-phase transactions[8] guarantees that the transactions here will still act atomically by virtue of first acquiring all of their resources (i.e. changing

all of its squares to white, effectively locking them), and then relinquishing them (i.e. giving them a color). Although the theorem does not rule out deadlock, the semantics here do so by ensuring that only one transition is initiated at any one time. It could also be ensured by enclosing initiation in a critical section covered by a lock, or by changing the color of the tape squares to white in a predetermined global order.

Predictable heads provide ample opportunity for optimization. Since only `read` and `nodata` heads are predictable (by definition), the new data state and control state for the associated square is known the instant the transition fires. This means that the scheduler itself can pre-`give` the square a color during scheduling and thereby immediately schedule other transitions which become ready as a result.

The control state (color) of each square is implemented as internal state which allows a scheduler to determine which transitions to schedule. It is often most efficient to distribute this control state among the transitions. That is, rather than representing the control state of each square in a particular location, each transition is given a *reasons count*—i.e. an integer which describes the number of the transition's heads which are over the wrong color of square. Each time a transition is initiated or executes a `give` statement, the appropriate reasons counts are adjusted (within a critical section), and new transitions are scheduled (i.e. initiated) whenever their reasons counts reach zero[7].

In distributed memory environments, `give` statements associated with `write` or `read-write` heads often map straightforwardly into message `send`s which pass the data state associated with the square to the next process that will read it. There are, however, some circumstances where the next process to read the data cannot be immediately known. In that case, the data state can either be left with the transition to be requested later when the reader is finally determined by the scheduler, or it can be forwarded to a location which is physically closer to all potential readers.

A simple subroutine package, called Cooperative Data Sharing (CDS), embodies the communication requirements of F-Nets without the dataflow-like execution semantics[4]. In addition to serving as a basis for the implementation of F-Nets, it also serves as a standard communication substrate for a variety of other purposes, similar to PVM or MPI except that it avoids all copying in low-latency shared environments.

3.6. Tolerance and Other Properties of F-Nets

The visual nature of F-Nets springs from the nature of computation and the relationship between algorithms and computations. In the sequential world, a computation is usually defined as a sequence of operations. One possible algorithm to express a particular sequential computation is a "straight-line" algorithm which performs each of the operations in the proper order, but the power of programming is obtained by "folding up" and compacting this straight-line algorithm with loops and conditionals. During execution, such a "folded up" algorithm both performs the operations designated therein and unfolds into a sequence at the same time, and the unfolding itself can be affected by the inputs provided to the algorithm.

Similarly, a parallel computation is often considered as a partial ordering of operations [11]. However, the term "parallel algorithm" has heretofore not had a very formal definition. An F-Net algorithm is an almost perfect analog to a sequential algorithm—i.e. it is a folded up partial ordering of operations, which is unfolded as it is executed. This "folded partial ordering" description explains why F-Nets are represented most naturally as graphs.

Unlike sequential algorithms, some F-Net algorithms may unfold into different partial orderings, even when given the same inputs (or in this case, initial markings). This nondeterminism is a desirable characteristic, as described earlier under semantic tolerance, as long as it is not introduced by accident. Potential non-determinism in an F-Net can be detected syntactically, so tools can allow the user to verify that it is desired. Specifically, an F-Net may be non-deterministic if and only if it contains two (or more) transitions which have like-color heads on the same square (say *s1*) and those same transitions do not have differing-color heads on another "shared" square (say *s2*). The non-deterministic choices made during execution can be recorded efficiently by just recording the order in which the stated transitions fire—i.e. one bit recorded for each execution of the offending transitions—and this information can be used during debugging to ensure repeatability.

Language tolerance is achieved in F-Nets because the model requires only that each firing function represent a deterministic mapping from some set of data values to some new set of data values and to a color for each head. The representation of this mapping is not restricted: e.g. it can be in the form of an imperative subroutine, as described, or in terms of a functional, dataflow, or logic program fragment (though these paradigms may be unable to take advantage of F-Nets' update-in-place capabilities to gain maximum efficiency). This provides maximum flexibility to use any language, and even to use different languages for different transitions.

The fact that each transition represents a simple mapping, independent of anything else going on at the time, is indicated by the (first) semantics. That is, even though transitions may execute concurrently, they must act as though they are executing one by one. This not only provides concurrency tolerance, since the constructs being used have identical behavior in parallel and sequential environments,

but also bug tolerance, since errors in implementing a transition can only lead to errors in the mapping represented by that transition. Moreover, since the semantics of the language used to implement the transitions does not significantly change in a parallel environment, standard sequential debugging tools can be used to debug the transition mappings. This is in marked contrast to traditional shared-memory or message-passing programming, where the behavior of any one program or program fragment can only be described by including the possible asynchronous arrival of messages and/or data, and therefore by including all possible global states of the system.

F-Nets achieve latency tolerance through all of the techniques mentioned in the previous section. Since each transition is endowed with the knowledge of the data that it will need to perform its task, this data can be forwarded (staged) by the scheduler in a dataflow fashion to the processor which will execute the firing function, even before the function executes. Latency is amortized by communicating an entire tape square (which could comprise a large data structure) at a time, leaving fine-grain access to its components to occur in a low-latency environment.

Queuing of multiple versions of a tape square is also supported by the model, due to predictable heads. Suppose that one transition has a green "predictable red" `read` head on a tape square, and another has a red `write` head on the same square which changes the color to green. When the first transition fires, the scheduler can immediately change the square color to red, allowing the writing transition to execute again even while the reader continues to execute. Of course, in this case, the scheduler must ensure that the writer uses a separate memory area to create the "next" version of the data state for the tape square.

Transitions are very tolerant to both concurrency and dynamic environment considerations due to their atomic, stateless properties. Specifically, by ensuring that each transition is relatively small, an algorithm can expand into any number of available processors, and problems related to the loss or migration of execution state can be avoided by backing out of partially-executed transitions. Nevertheless, unlike some functional and dataflow models where data must be copied from one actor (function, program, chare, etc.) to the next, executing multiple sequentially-composed transitions on the same processor adds virtually no overhead above a standard subroutine-call interface.

4. Software Cabling

Software Cabling (SC) is a visual programming language for building very large F-Nets. An SC program effectively compiles into an F-Net while ensuring that the correspondence between the SC program and the F-Net is always apparent. This allows SC to inherit many of the desirable properties of the F-Nets model while compensating for some of F-Net's apparent practical deficiencies.

The terminology which is used to describe SC is based upon a hardware analogy. In the first subsection, terminology and representation of the basic constructs will be described. Subsequent subsections will describe modularization, first-class modules, objects, templates, arrays, and data parallelism.

4.1. Basics

An SC program is constructed of *modules*. Each module has a *body*, which tells it how to act, and an *interface*, through which it interacts with its environment. The interface consists of one or more *pins*, each with a name, a permission (`read`, `write`, `read-write`, or `nodata`), a data type, and a list of identifiers called a *signal set*. There are two kinds of modules: *chips* and *boards*.

A chip can be considered as a custom CPU chip, and is the only construct in SC which transforms data. As with an F-Net transition, its body is constructed with tools outside the realm of SC, but SC depends upon each chip acting in a specific deterministic way when initiated: The chip must initialize all of its internal data, then read one data item from each of its read or read-write pins, then compute for some amount of time and write one data item to each of its write and read-write pins. In addition, for each of its pins, it must post a signal from the signal set of that pin. (If a signal is not posted for a pin, a special "bottom" signal is imagined to be posted.)

A board can be considered as a flat surface upon which other components are mounted. The body of a board is specified in a schematic-like diagram. The primary role of a board is to specify how other modules which are mounted upon it will interact.

The two basic components which are permanently mounted upon boards are called *sockets* and *memories*, and these are connected by *wires*.

A socket is represented in the board diagram as a circle, and is designed to hold a module—i.e. either a chip or a board. To this end, a socket consists of a set of receptacles, each capable of accepting one pin of the module interface. Each receptacle has a name, a permission, a data type, and a signal set—just like a pin—and will only accept a pin of the same name, signal set, and (for simplicity) datatype. The receptacle's permissions can typically be more general than that of the pin, as shown in table 2. The socket circle is labeled with the name of the module inserted into it.

Receptacle permissions	Accepts pin with permissions
read-write	read-write,read,write,nodata
read	read, nodata
write	write
nodata	nodata

Table 2: Kinds of pins accepted in a receptacle

A memory is represented in the board diagram as a rectangle, and is designed to hold one data item and one color, both of which may change during execution. The type of data item which a memory may hold is described by a data type associated with the memory. The initial data state of the memory is initialized to standard default values, which can be over-ridden by annotating the memory rectangle with "`=`*`const`*" where *`const`* is a constant—i.e. a literal numeric, character, or record constant in a standard form, or of the form "`<`*`name`*`>`" in which case the constant associated with identifier "*`name`*" is found in a repository associated with the program (called the *program data base*).

Wires are represented in the board diagram as colored lines, and they connect each of the receptacles of a socket to a (different) memory. Each line is labeled with the name of the receptacle which it connects, and has arrowheads to represent the permissions of that receptacle: on the socket end for `read`, the memory end for `write`, both ends for `read-write`, and neither end for `nodata`. The signal set of the receptacle is represented just inside of the memory rectangle, where the wire connects to it, and each of the identifiers in the signal set is colored.

If chips are placed into all of the sockets on a board, the semantics are almost exactly those of the F-Net with the same appearance. The memories act as tape squares, the wires as heads, the sockets as transitions, and the chips as firing functions. The only difference is that a chip now posts a signal to each pin instead of providing a color, and the new color of the memory is determined by the color of that signal name in the signal set within the memory. (Bottom signals always result in colorless memories.) This effectively parameterizes the colors, resulting in the ability to use a single chip in more varied circumstances than a similar firing function

Table 2 was designed specifically to guarantee that SC programs appear as F-Nets, which is the reason that a `write` receptacle cannot accept a `nodata` pin. If otherwise, it would be possible to run an experiment that did not match that of an F-Net—specifically, initializing all of the memories for a socket, letting the chip therein execute, then re-initializing the memories to the same values again but changing the value for one attached to a `write` receptacle, and letting the chip execute again. The memory on the `write` receptacle should end up with the same value both times, but wouldn't if a `nodata` pin was inserted into the receptacle.

4.2. Modularization

So far, there has been no mention of how a board's interface is connected to its body. This is accomplished through special memories called *i-memories* (interface memories). Specifically, for each pin in the board's interface, there is one i-memory in the board, labeled with the name of the pin. The signals from the pin are listed in a small box, called a *posting table*, which adjoins the i-memory rectangle. Each signal name in the posting table has a different color, and none of them are green.

To describe the result of inserting a board into a socket, it will help to first define some terminology. If board A is inserted into a socket on board B, board A is called the *primary* board, and board B the *secondary* board. Since each i-memory on board A corresponds to a pin of board A, and each pin of board A also corresponds to a receptacle on board B which is connected by a wire to a memory on board B, there is a correspondence between each i-memory on board A and a wire and a memory on board B. These wires and memories on board B are called the respective *targets* for the i-memories on board A.

The semantics are as follows: In all cases, an i-memory effectively shares the data state of its target memory. The control state (i.e. color) of each i-memory is initialized to be colorless, but whenever the target memory of an i-memory matches the color of the target wire, the i-memory itself becomes green. If a module on the primary board (with a green wire) accesses that i-memory, it not only steals the color from the i-memory, but also from the secondary memory. When (if) that module posts a signal to give the i-memory a new color, SC compares the new color with the colors of the signals in the posting table. If there is no matching signal color, the i-memory retains the new color (as usual), and other modules on the board which are attached to the i-memory with wires of that color may fire as usual, but the target memory remains colorless. However, if a signal color matches, the i-memory becomes colorless once again, and the matching signal is effectively posted through the target wire to the target memory.

4.3. First-Class Modules

The modules (i.e. chips and boards) defined previously are not actually immutable objects. Instead, they are descriptors of immutable objects. That is, the user provides a description of the module to create, and the socket actually creates it. These descriptors are constants, stored in the program data base along with any other constants the user wants to store there. In the cases described above, this subtle difference was not important because each socket was always given the same descriptor, and each board which was created remained intact for the remainder of the program. This subsection describes other cases, where modules are treated as first-class objects.

Each socket has a special receptacle called `(module)` from which it reads the descriptor for the module that it is to create. The labeled-circle representation used in the previous section is actually a shorthand for this: see figure 4.

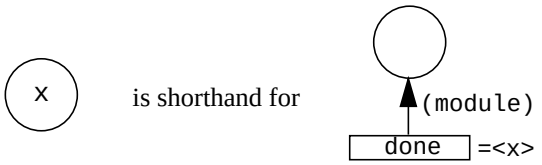


Figure 4. Labeled circle shorthand

When the wire for the `(module)` receptacle is the same color as the memory to which it is attached (which it will always be in the default case above since the `done` signal and the wire are green), the socket reads that descriptor. If the descriptor is that of a chip, the socket builds the chip and waits for it to become ready, then initiates it. If the descriptor is that of a board, the socket associates the i-memories on the board with their target memories on the secondary board as discussed in the last section and creates and initializes all of the other memories on the board (unless they have been previously created, as will be described in the next section). This may, in turn, allow other sockets to construct their modules—i.e. if a newly-created memory (or a newly associated green i-memory) on the board is attached with a green wire to a `(module)` receptacle for a socket on the board.

The `(module)` receptacle does not act exactly like other receptacles. Even when the socket reads the descriptor on its module receptacle, it does not actually drain the color from the associated memory until and unless the resulting module actually has some outward effect on other memories—i.e. drains the color through one of the other receptacles of the socket. If another socket changes the color of the memory containing the module before the socket has its effect, the socket must back out and effectively pretend that it never tried to execute. Note also that the `(module)` receptacle is predictable by default, and a socket will never re-start until the module therein finishes.

By default, boards never finish, but a board is permitted to have an i-memory named `(module)`, and if so, this i-memory naturally corresponds to a `(module)` pin on the board. Changing its color to match a signal in its posting table will have the natural effect of causing the socket containing the board module to post that signal to its `(module)` receptacle. It will also have one other side-effect: From that point on, no i-memory for the board will ever be green again. In other words, if some of the i-memories on the board are a non-green color at that time, they will continue to possess that color, and if modules on the board have stolen color from some i-memories on the board, they are free to post signals and return the color to those i-memories, but if an i-memory is (or becomes) colorless because its target memory is the wrong color, then the i-memory will remain colorless from that point on. Once the color on all of the i-memories of the primary board are effectively stolen back by the secondary board, the primary board will have no further effect on its environment, even if other modules on the board are able to execute, so the board can be "garbage collected" by the runtime system.

A board can be (and is) considered to have logically finished as soon as it has posted a signal to its `(module)` pin, since this allows the board to finish up what it started, but does not allow it to start more work. In atomic transaction terminology, these semantics ensure that the board is entering a shrinking phase. If the programmer ensures that the board does not enter a second growing phase (i.e. does not steal color a second time) before posting this signal, the board execution can be assured to be an atomic transaction.

A special chip called `copy`, provided by SC, increases the power of this feature, along with being useful in other contexts. The `copy` chip has two pins: a `read` pin called `in`, and a `write` pin called `out`. Its operation is obvious: it reads a data item from its `in` pin and writes the data item to its `out` pin. It is better than a user-written copy chip, in that it can copy any kind of data object, including module descriptors, and it can do so very efficiently (perhaps without even performing an extra copy) because the SC scheduler knows the desired result and can therefore optimize.

4.4. Objects

A board descriptor can be considered as an abstract data type, with each socket containing that board being an instance of that type, since a board has fixed interfaces, methods (i.e. modules within attached to i-memories with green wires) which can be invoked by outside actions, and hidden data (i.e. memories) to which they control access. But funneling all accesses for one instance through one socket is extremely cumbersome. What is needed is for a single object to be usable at different places within a network in a concurrent fashion. The first-class modules described in the last section set the stage for this. This section finishes the job.

Each non-interface memory on each board has an immutable *instantiation level*, which is 0 by default but can be specified as some positive integer by the programmer, shown graphically as a number of lines parallel to one or more sides of the memory rectangle. The board itself also has an instantiation level, which is initially set to the maximum instantiation level of any memory on the board.

These instantiation levels are used by a special chip, provided by SC, called `instant`, which has a single `read-write` pin called `object` and a datatype of "module" (i.e. a module descriptor). When an instant chip fires, it reads the board module from its `object` pin and finds all of the memories on the board with the same instantiation level as the board. It then *instantiates* these memories, decrements the instantiation level on the board descriptor, and writes the new board descriptor back out to its pin.

To relieve any confusion between instantiated boards

and non-instantiated board descriptors resulting from the above paragraph, consider that the control and data states for all memories are kept in a special globally-accessible data area called the memory heap. Instantiation then just corresponds to creation of a memory in the memory heap and saving the address of this new memory with the memory rectangle as part of the new (constant) board descriptor.

The constructs described here can be used to facilitate object-oriented programming. First, the programmer assigns an instantiation level of 1 to "instance variable" memories, resulting in a board descriptor with an instantiation level of 1 which serves as a class. To create an object of that class, the user copies the class to another memory (using `copy`) and instantiates it (using `instant`), which creates the instance variables and upgrades the class descriptor to an object descriptor. Any socket which reads this object descriptor will share the control and data state for the instance variable memories. Instantiation levels of 2 and greater can be used to repeat this approach in a hierarchical fashion (e.g. for class and superclass variables).

4.5. Patterns and Templates

Programming-in-the-large requires the construction of templates or skeletons which describe the interactions of individual modules or objects without over-constraining the form of those modules. The constructs in this subsection help to address some of those requirements.

I-memories, wires, pins, and receptacles are all simple atomic constructs. These are actually the degenerate forms of record-like constructs, called *i-sets*, *cables*, *pin-sets*, and *receptacle-sets*, respectively, which can fill the same roles. So, an i-set is recursively defined as a set of i-sets or a single i-memory; a cable as a set of cables or a single wire; a pin-set as a set of pin-sets or a single pin; and a receptacle-set as a set of receptacle-sets or a single receptacle. The power of these constructs is that the programmer does not need to fully specify their form: unspecified portions of the hierarchy are inferred (dynamically) from the current execution context.

An i-set is shown graphically as a labeled rectangular region, as in figure 5. If it is not otherwise apparent, the region can be distinguished from a memory rectangle by being drawn with a dashed line. It encompasses any i-sets (including i-memories) which it contains. An asterisk within an i-set represents zero or more additional unspecified i-sets. An empty i-set rectangle is a special case which represents either an i-memory or an i-set containing any number of component i-sets. An imaginary i-set, called `(iface)`, encompasses all other i-sets and i-memories.

Just as there is a one-to-one correspondence between i-memories on a board and pins in the board's interface, there is the same correspondence between the i-sets on the

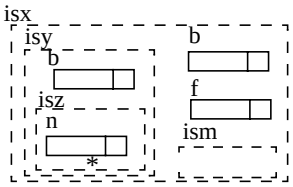


Figure 5. I-sets. isx={isy={b,isz={n,*}},b,f,ism=?}

board and pin-sets in the board's interface. And, just as pins on a board fit into receptacles of a socket, pin-sets from a board fit into receptacle-sets of a socket. It is at this stage where all unspecified structures become known to SC. That is, when a board interface is inserted into a socket, that interface is made to conform to the socket, defining any unspecified portions of any pin-sets in the interface which in turn defines any unspecified portions of any i-sets on the board.

Cables, which are shown as lines (again, dashed if necessary to distinguish them from wires), connect the receptacle-sets of a socket to other entities, and thereby the receptacles within those sets to memories. The simplest such binding is shown by drawing a cable between the socket and an i-set, in which case the form of the cable (and therefore the receptacle-set with the same name in the socket) is inferred to have the same structure as the i-set. Each wire of the cable (i.e. receptacle of the socket) is attached to the i-memory of the i-set with the same name, and with a signal set identical (in color and name) to the posting set of the i-memory.

Figure 6 shows that cables can also be built up from individual cables (including wires) using the *bundling* construct, shown as a triangle. The cable being built is attached to the apex of the triangle, and the component cables are attached to its base. This construct is unidirectional—the cable from the apex must always connect to a socket or the base of another bundling.

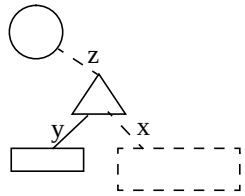


Figure 6. Bundling wire y and cable x into cable z

With these record-like constructs, SC programmers can build a "template" board, which reads objects or modules from some of its pins and specifies how they should interact to a limited degree, even if the entire interface of those modules is not known. Likewise, a module can access part of its interface without necessarily knowing the form of the entire receptacle-set into which it is inserted. For example, a socket could pass a hierarchical file system to a module by representing each directory or folder as a receptacle set and each file as an receptacle, and a module inserted into the socket needs only to identify the specific items which it

knows about within the corresponding i-set.

4.6. Arrays

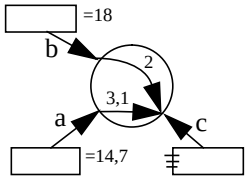
Memories within SC, like tape squares within F-Nets, can only be accessed by a single chip at a time. This is especially troublesome for arrays, since storing the entire array in a single memory would greatly restrict parallelism, but storing each element in a separate memory can be completely impractical. Neither approach lends itself well to arrays which change their size dynamically during program execution, making data parallelism difficult or impossible to express. Dynamic memory allocation is also desirable.

To address these problems, each memory in SC has an associated *dimensionality*, expressed numerically as a non-negative integer and graphically as a number of hash marks in the left side of the memory rectangle. Dimensionality is basically a number of dimensions, and memories with dimensionality of zero, like all the memories discussed so far, are sometimes called *scalars*, while those with other dimensionality are called *arrays*. Each array consists of an infinite number of elements, and all elements have the same type and same initial data state, but each element maintains its own control state and data state during execution. Each element is uniquely addressed by an index which consists of *n* integers, where *n* is the dimensionality of the array. Since i-memories can be arrays, so necessarily can pins, receptacles, and wires. An array pin conforms to the size and shape of its array receptacle at insertion.

To understand how memory array elements are accessed requires further explanation of sockets. In a previous section, sockets were described as waiting until their `(module)` receptacle was ready, then snooping on the memory attached to that receptacle to determine the module to execute. In fact, sockets can have many *phases* (i.e. levels of snooping), of which reading the `(module)` receptacle and building and executing the module is the last. Each other phase waits on and reads one or more *primary* receptacles and uses the information there to establish the size in each dimension of one or more *secondary* receptacles, and to bind (i.e. connect) wires from the secondary receptacle elements to memory array elements. A secondary receptacle from one phase can be used as the primary receptacle of a subsequent phase. Just as before, if any memory attached to a primary receptacle is accessed by another module before the socket finishes all of its phases and the new module therein has some outward effect, then all phases effectively start all over again from scratch. This rule ensures that the socket binding and execution of the module therein appear as an atomic action.

There are two basic kinds of *binding modifiers*, called *selection* and *translation*. These are shown as an arrow within the module circle, from a wire representing the primary receptacle (to be snooped) to a wire representing the secondary receptacle (to be accessed using the snooped data). Each arrow is labeled with an *index list* (i.e. a list of small integers), which are prefixed with a "+" if indicating a translation.

A selection is very similar to normal subscripting. When the primary receptacle becomes ready, *n* integers are read from it, where *n* is the length of the index list on the arrow, and they are used, in order, as indices into the primary receptacle, as specified by the index list. Put another way, each integer effectively collapses the primary receptacle in one dimension, specified in the index list. Multiple selections can be specified for the same secondary receptacle as long as their index lists do not contain the same values. Specifying all of the indices for the secondary receptacle, as in figure 7, collapses it to a single element, like normal subscripting. Leaving some indices unspecified results in a secondary receptacle of reduced, but non-zero, dimensionality. This is not acceptable if the module inserted into the socket in the final phase is a chip, since a chip can access only a finite number of elements, but is acceptable if the module is a board, since further selections can be performed on the associated i-memory within the board.



Result: c is scalar receptacle, bound to element 7,18,14 of 3-dimensional array

Figure 7. Selection bindings: scalar from 3 dim.

Any number of indices in the index list for a selection can be enclosed in parentheses, in which case two integers (rather than one) are read from the primary receptacle for that index. Those integers are used as the bottom and top of an index range. Note that such ranges do not reduce the dimensionality of the secondary receptacle, but simply limit the number of elements in the specified dimensions. These range selections are so useful that SC provides a shorthand for their use, as in figure 8: By attaching one end of a wire to the corner of a scalar memory having a datatype of two integers, and the other end to one of the dimensionality hash marks of a memory rectangle, a range selection will be performed for the index corresponding to the hash mark for all subsequent accesses to the array memory. If the array is an i-memory, a similar notation (with reversed arrow) initializes the range memory with the size of the socket receptacle in that dimension.

The above description of selections refers to the special case when the primary receptacle is scalar. In the general case, a selection is performed as above using each element of the (finite) primary receptacle, and the results are organized into an array shaped just like the primary receptacle. The dimensionality of the result is therefore effectively

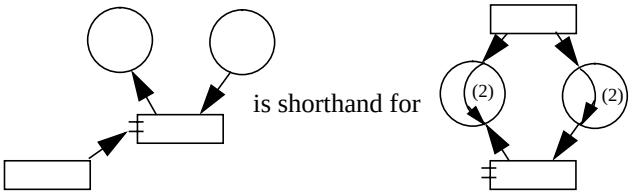


Figure 8. Shorthand for range selections

increased by the dimensionality of the primary receptacle, just as it is decreased by the length of the index list. Ranges are not allowed with non-scalar primary receptacles.

Like selections, translations read *n* integers from the primary receptacle, but instead of using them as indices into the secondary receptacle, they are used as offsets for those indices. This allows the secondary receptacle to be logically shifted in any direction by any offset, being especially useful with a selection representing a stencil.

4.7. Data Parallelism

To provide data parallelism, a language needs not only to support arrays, but also a way to scale the parallelism with the size of the arrays. For this purpose, SC provides the DupAll and DupAny constructs, which also execute as phases.

The DupAll is only permitted for receptacles connected to memories having a datatype of two integers, and is represented by prefixing the receptacle's name with an asterisk. When the receptacle becomes ready, SC reads the two integers from the memory and treats them as the bottom and top of a range. It then effectively "clones" the socket to produce one socket for each number in the range, leaving all of the wires and receptacles alone except for the DupAll receptacle. For each clone, this receptacle's type is changed to integer, the asterisk is removed from its name, and the receptacle is wired to a separate new integer memory which is created specifically for that clone by SC and initialized to a unique integer from the range.

Each of these new cloned sockets persists only long enough to execute one module until it finishes. When they all finish, a `done` signal is posted to the original memory, after which the DupAll may do it's job again. The DupAny is exactly the same as the DupAll except that (1) a "+" prefix is used instead of a "*", and (2) the module in only one of the sockets will be allowed to execute before the `done` signal is posted and the operation is reset.

DupAll receptacles are often used as primary receptacles for selections or translations, allowing a single socket to be replicated for each element (or dimension) of an array. It is common to use a separate DupAll for each dimension.

5. Conclusion

Tolerant programming is possible. Given a satisfactory theoretical model as a basis, many of the difficulties related to parallel programming can be surmounted. F-Nets provides a formal and natural expression for parallel (and sequential) algorithms, and can serve as a basis for constructing tolerant programs in languages like SC.

6. Acknowledgments

The idea of tolerant programming benefited from discussions with colleagues Robert Hood and Louis Lopez at NASA Ames. The F-Nets model was developed during studies at Oregon Graduate Institute, where it began with the research and helpful feedback of my advisor Robert Babb, and was furthered by feedback from my committee (Michael Wolfe, Harry Jordan, and Richard Kieburtz) and early discussions with Dick Hamlet. The CDS package was developed at NASA Ames Research Center, where support was provided by the NAS program and HPCC. The SC presentation benefited from discussions with Doreen Cheng at NAS.

References

1. R. G. Babb and D. C. DiNucci, "Design and implementation of parallel algorithms with Large-Grain Data Flow", in The Characteristics of Parallel Algorithms, Jamieson and Douglass (ed.), Cambridge, MA, MIT Press, 1987, pp. 335-349.
2. K. M. Chandy and J. Misra, "Parallel Program Design: A Foundation", Reading, MA, Addision-Wesley, 1988.
3. D. C. DiNucci, "A simple and efficient process and communication abstraction for network operating systems", LNCS vol. 1199 (CANPC'97 Proceedings), pp.31-45, Berlin, Springer-Verlag, 1997, pp. 31-45.
4. D. C. DiNucci, "CDS", http://www.nas.nasa.gov/Tools/CDS
5. D. C. DiNucci, "A formal model for architecture-independent parallel software engineering", Ph.D. Dissertation, Oregon Graduate Institute, 1991, also available at ftp://ftp.netcom.com/pub/di/dinucci/thesis.ps.Z
6. D. C. DiNucci and R. G. Babb II, "Design and implementation of parallel programs with LGDF2", COMPCON'89, San Francisco, 1989, pp. 102-107.
7. D. C. DiNucci and R. G. Babb II, "Practical support for parallel programming", Proc. 21st HICSS Software Track, 1988, IEEE, 109-118.
8. K. P. Eswaran et al, "The notions of consistency and predicate locks in a database system", CACM, vol. 19, 11 (November 1976), pp. 624-633.
9. R. Milner, "A calculus of communicating systems", LNCS, vol. 92, Berlin, Springer-Verlag, 1980.
10. J. Peterson, "Petri net theory and the modeling of systems", Englewood Cliffs, NJ, Prentice-Hall, 1981.
11. V. R. Pratt, "Modeling concurrency with partial orders", International Journal of Parallel Programming, vol. 15, 1(February 1986), pp. 33-71.
12. T. vonEicken, "U-Net: A user-level network interface for parallel and distributed computing", ACM Symp. on Oper. System Princ., Copper Mountain, CO, Dec. 1995, pp. 303-316.