

F-Nets and Software Cabling: Deriving a Formal Model and Language for Portable Parallel Programming

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

Abstract - Parallel programming is still being based upon antiquated sequence-based definitions of the terms “algorithm” and “computation”, resulting in programs which are architecture dependent and difficult to design and analyze. By focusing on obstacles inherent in existing practice, a more portable model is derived here, which is then formalized into a model called F-Nets which utilizes a combination of imperative and functional styles. This formalization suggests more general notions of algorithm and computation, as well as insights into the meaning of structured programming in a parallel setting. To illustrate how these principles can be applied, a very-high-level graphical architecture-independent parallel language, called Software Cabling, is described, with many of the features normally expected from today’s computer languages (e.g. data abstraction, data parallelism, and object-based programming constructs).

I. Introduction

In 1936, Alan Turing proposed a formal model for computation, and the notion that computation and Turing Machines are inextricably linked has since been widely accepted, often described as the Church-Turing Thesis. Since all of the important action in a Turing Machine computation occurs at a single point (under the machine’s single “head”), the space-time diagram of the computation is naturally a line—i.e. a sequence of operations. Although programs for traditional computers don’t look much like programs for Turing Machines, this perception of a computation as a sequence of operations has had a pervasive effect on how we program virtually all computers. An algorithm, or more specifically, a program, is a description (usually abbreviated or compressed) of the computation to be performed, so it should be no surprise that algorithms typically resemble compressed or folded sequences of operations.

One might imagine that the development of parallel programs would begin with the development of parallel algorithms, but the history of the term algorithm as a means of expressing a sequence of operations has made the term “parallel algorithm” an oxymoron. Instead of correcting this by extending the term’s definition to encompass its new requirements, the common approach to parallel programming has

been to focus on the fact that almost all parallel computers are constructed from a set of interconnected standard sequential processors. This has led us to dispense with the notion that a program is the manifestation of a single algorithm, and instead to consider a parallel program as a set of sequential programs (or processes or objects) which communicate, each being the embodiment of a sequential algorithm. This view of parallel programming has been particularly troublesome, for a number of reasons which will be discussed.

Functional and dataflow programming approaches, based more closely on Church’s recursive function theory, have been offered as alternatives to this sequential thinking. Unfortunately, these notations, where the program is expressed as a function, rob the programmer of an extremely powerful semantic construct which is efficiently implementable on virtually every commercial computer—i.e. volatile memory. As a result, the programmer must either go to extremes to make do without memory, often by copying data superfluously (e.g. to mimic data structure update), or the language must bend the rules and allow some cases of volatile memory semantics, thwarting its functional properties. These approaches also often force compilers, rather than programmers, to identify sensible opportunities for parallelism, and language processors and other tools which are responsible for efficiently mapping the functional semantics back onto the underlying hardware have not demonstrated that they are up to the task—i.e. they have only rarely competed with imperative sequential languages. In addition, the functional models, by definition, cannot express non-determinism, and important issues like I/O do not fit naturally within the model.

Other approaches have also had limited success. Data-parallel, or SIMD-style, languages (e.g. HPF) are based on a model where the programmer envisions a single flow of control, which is multiplied (where possible) to work on multiple data items at once, and its application is therefore limited to cases which fit that model. CSP-based languages (e.g. Occam) are based on small-granularity approaches where traditional sequential languages are disposed of completely, and these have proven viable only on processors designed specifically to facilitate them. The Bulk-Synchronous Parallel (BSP) method introduces synchrony as a basis for simplifying programs, but requires compilers and runtime systems to try to

remap to an asynchronous style for performance. Further comparisons will be made after explaining the current model.

This document proposes going back to the source of the problem—i.e. the definition of the terms “algorithm” and “computation”. It begins in section II with an analysis of the drawbacks of the communicating process approach, and then in section III describes a step-by-step approach to remedy some of these drawbacks. As a result, in section IV, a new simple, formal, and portable computational model, called F-Nets, is proposed, which suggests new more general definitions for “algorithm” and “computation” which encompass both sequential and parallel notions. In section V, these definitions are compared with the more traditional ones, leading to insights such as a more general definition of structured programming, and some properties of the F-Net model are explored. Finally, in section VI, these notions are then used to motivate a new high-level architecture-independent parallel programming language, Software Cabling.

II. Problem Statement

Before embarking on solutions to problems arising from traditional parallel programming approaches based on communicating sequential processes, it is important to understand what those problems are. Many of these problems originate with the fact that communication is the secondary result of operations in the individual processes rather than being specified directly. That is, the programmer uses individual operations within separate sequential algorithms to hopefully cooperate to move the data between the intended algorithms in the intended manner at the intended times. Tools are generally not available (and in many cases, cannot be constructed) to statically (i.e. syntactically) determine whether or not the programmers intentions have been correctly encoded.

Non-determinism, a natural and useful property of parallel computations, is not even expressible in sequential algorithms (or within a Turing Machine). It can be introduced between processes in a parallel program, but only by using an approach identical to an improperly-constructed communication. Because of this, and because of the difficulty in finding non-determinism which has been unintentionally introduced (so-called “Heisenbugs”), non-determinism has become scorned by parallel programmers, rather than being regarded as one more tool to speed programs and to simplify programming.

Whereas a sequential algorithm can be implemented in just about any computer language, and is portable to nearly any sequential computer, a set of communicating sequential algorithms is only as portable as the communication mechanism. Specifically, multi-processor computer systems are typically built to optimize one communication mechanisms at the expense of others, so parallel programmers immediately restrict themselves to a particular class of target architecture the moment they choose a communication mechanism. Simi-

larly, they further restrict themselves to specific sizes and types of machines depending upon the granularity at which communication is performed.

Because the programmer is required to specify the order (i.e. sequence) in which operations execute on each processor, and because the individual processes execute asynchronously, communication often requires one process to stop and wait for data from another to become available, leading to potentially wasted CPU cycles. To remedy this, parallel programs are often (manually) tuned, based on specific environmental assumptions, to communicate at specific intervals, but this takes a great deal of planning, experimentation, and/or knowledge of both computation and communication timings. As a result, even if the programmer can succeed at minimizing execution time under some particular circumstances, any change in the computation or communication timings resulting from unexpected loads on some system resources or from executing the program in a different environment can obviate all such planning.

In spite of such important and formal studies as Hoare’s CSP (Communicating Sequential Processes) and Milner’s CCS (Calculus of Communicating Systems), the addition of communication between processes adds immense complexity and makes a parallel programs extremely difficult to analyze formally. Knowing the input-output mapping which each process expresses is not sufficient to determine the behavior of those processes when composed into a parallel program—the relative order in which those inputs and outputs are consumed or produced must also be taken into account. Even after a parallel program has executed, determining the function which it computed is extremely difficult, because each communication effectively transfers a portion of the internal state of one process to another, and since the processes are typically long-lived, this internal state can be the result of an arbitrarily-long history of computation within the process initiating the communication (and preceding communications between that process and others).

III. Derivation of a new model

Although the development of a new computational model can have many different starting points, we justify some features here by beginning with traditional programming practice—i.e. shared-memory and message-passing programming—and then performing stepwise modifications to introduce a more portable (though compatible) communication model, and a process model which lends itself to more abstract and processor-independent specification.

A. Portable communication model

For a communication model, we wish to gain portability by preserving the advantages of both message passing and shared memory, so we begin by identifying their similarities. For

example, both a `send` primitive in a message-passing semantics and an `unlock` primitive in a shared-memory semantics can be considered as an act of making something available. In fact, if we temporarily make a simplifying assumption that each shared-memory lock is statically associated with some region of shared memory which cannot be accessed without first exclusively acquiring the lock, then both of these primitives (`send` and `unlock`) effectively make a specific data item available. (Non-exclusive locks for reading will be considered later.) These primitives also provide an identifier with which the data item can eventually be found: For `send`, this identifier is typically a process ID and message tag, and for `unlock`, the identifier is the address of the lock. More abstractly, we call this act of making data available “tossing” the data item.

Likewise, both a message `receive` and a shared-memory `lock` primitive can be considered as finding an available item with a given identifier, gaining access to the item, and making it unavailable. (This makes the simplifying assumption that, for shared memory, every item has been implicitly `tossed` at the beginning of execution). This operation will here be called “catching” an item.

This is somewhat of an over-simplification. For example, in message passing, one can usually expect to `toss` (i.e. `send`) many items with the same identifier (i.e. process and tag), and to have them queued or otherwise buffered so that all can eventually satisfy `catches` (i.e. `receives`), but in shared memory, one usually expects each `toss` (i.e. `unlock`) for a specific identifier to effectively over-ride previous ones. This difference in semantics will be remedied by simply forbidding a process from `tossing` an item if there is already an item with the same identifier outstanding (i.e. waiting for a `catch`). Such a restriction is not a major imposition for shared memory, since there are no convincing reasons that a program must be able to perform multiple `unlocks` without intervening `locks` for the same data item, but more consideration is given here to restricting message buffering to only one unreceived message per tag/process at a time.

Buffering serves primarily as a means of compensating for long latencies between processes, by allowing the receiving process to accept data from the sender while the receiver is still performing other work, thereby effectively hiding the latency of the communication. Rarely is more than a single buffer for each process/tag combination used to accomplish this. Although more levels of buffering can theoretically allow the “downstream” process to get even further behind the “upstream” process without interference or failure, every system has some (perhaps dynamic) restriction on the amount of buffering available, so the program must somehow be made aware of this limit and must control (or “squench”) message production. The equivalent of such n -buffering can be provided with single-buffering-per-identifier by simply using n identifiers, and later in this document, even the constraint for a single identifier will be relaxed through optimization tech-

niques. It is also worth noting that the single buffering described here is somewhat more extensive than the complete lack of buffering guaranteed by the MPI standard for the `mpi_send` and `mpi_receive` primitives.

This description of identifiers and items has glossed over the fact that, for message passing, part of the identifier associated with an item—i.e. the process ID—does not just identify the item, but also identifies the process which is allowed to `catch` it. This is a valuable property in a high-latency environment, because an item can begin moving toward its target even before that process explicitly tries to `catch` it, thereby hiding latency. Still, shared memory semantics provides different benefits by not requiring that a single target be identified, in that demand-driven programming (e.g. where processes compete for data from a common pool or queue) is simplified. To provide the new model with the benefits of both message passing and shared memory, the identifier will now be broken into a *name*, which identifies the item, and a *target set*, which contains the names of processes which are allowed to `catch` the item. This target set augments the expressiveness of both shared-memory and message-passing communication. The contents of the set will be specified when the item is tossed, based on the needs of the algorithm rather than the underlying hardware support.

Summarizing, one can now imagine processes as executing in a sea of items, each of those items having a name and a set of potential targets. The items with only one potential target will benefit by gravitating toward that process to be ready if/when the `catch` primitive eventually executes. An operational description of what happens when a process `tosses` or `catches` an item—i.e. whether the data is copied to/from the process or simply accessed where it is—has been intentionally omitted, but a process which `catches` an item is assured to see it as it was when it was `tossed`. As a result, an observer watching processes `toss` and `catch` items will not be able to distinguish between shared memory and message passing, but the advantages of each has largely been preserved in this model.

B. Portable process model

The derivation of a process model will be driven primarily by the desire to make parallel program design scalable, which we define as the property that it be possible to analyze potential program behavior while limiting the scope of the analysis to a small number of processes at any one time. Traditional parallel programming does not generally have this property, in that an understanding of the possible behaviors of any process requires consideration of all possible interleavings of important (externally visible) events from that process with those from other processes, leading to a combinatorial explosion of possible interleavings which must be examined, made even worse by chains of such interactions which may be arbitrarily long.

To prevent such chains of interactions, the interleavings of externally-visible events in even two different process executions must be made unimportant. Put another way, it must be possible to reason about all of the operations (externally visible or not) within a process execution independent of other process executions. This is nearly equivalent to stating that any schedule of process executions has an equivalent sequential schedule—i.e. one in which only one process executes at a time—and this property is commonly known as execution atomicity. (Programs with this property have the added advantage that they do not depend on any certain number of processors, and can run on as few as one processor by serializing the process executions.) A long-known theorem from the database community[9] holds that a set of transactions (i.e. process executions) will have such a property if and only if each is “two-phase”, where a two-phase transaction is one which does not acquire any further shared resources once it has begun to divest itself of shared resources. In the case of our model, the most obvious shared resources are the items described in the last section, so a program will seemingly have the desired property if, for each process execution, all of the `toss` operations (if any) are preceded by all of the `catch` operations (if any).

In fact, even this step is not quite sufficient to achieve scalable parallel program design, because atomic transactions can still deadlock, and the order in which different processes `catch` their items may affect whether or not those processes can deadlock. That is, deadlock itself is a non-local property. This could be remedied by mandating a runtime system to watch for deadlocks and to “back out” (i.e. terminate) deadlocked transactions, but such complex runtime support is out of character. Instead, we insist that the processes not specify the order in which they will `catch` items, but that they instead just provide a set of the names of all of the items that will be needed, called a *catch set*. A process will not be initiated until and unless all items named in the catch set are caught. This also removes the need for an explicit `catch` primitive.

In addition to items being caught, there is another less obvious class of shared resource needed by these processes. Because of the restriction that a given name can be used for at most one item at a time, unused item names are also effectively shared resources, so a process must also ensure that names to be used for `tossing` items are available (i.e. unused). Rather than treating unused item names as a special case, this problem is solved by simply making items indestructible. In other words, even if a process just wants to `toss` an item, it must first `catch` an (empty) item with that name, by listing it in the catch set. Likewise, even if a process just wants to `catch` an item, it must later `toss` an (empty) item with the same name. Items which have not been explicitly `tossed` when a process finishes are automatically tossed.

These restrictions on process execution may seem to be rather extreme, troublesome, and inefficient. Programmers

may not feel comfortable with dividing process executions into two-phase transactions, or tossing empty data containers only so that they can be reused. The model may seem less burdensome, however, when one realizes that processes have become very similar to subroutines, with the catch set being analogous to the argument list. Like a subroutine, these processes have full access to all of their “arguments” (i.e. data items) when they begin execution, and can read or write those “arguments” during execution. Unlike a sequential subroutine, these processes can make their “arguments” available to other entities one at a time via `toss`, rather than keeping control of them all until completion. As each item is `tossed`, the process assigns it a set of potential targets, and a new process can start up whenever all of its arguments (i.e. items) are available to it. A process can iterate by simply targeting all of its items back to itself, thereby allowing itself to start up again immediately.

Carrying this analogy to subroutines a step further, we assign a *usage* to each item name in the catch set for each process. A usage is either: IN, which means that the process can read the data in the item but will not modify it; OUT, which means that the process can deposit data into the item, but will empty the item (if not already empty) when the process starts; INOUT, which means that the process can both read and write the data in the item; or NODATA, which means that the process cannot read or write the data in the item. (NODATA items are used only for their target sets, to control when processes can start.)

The conditions which must be true for a process to execute (also known as the safety condition) have been described—i.e. whenever, for all of the items named in the process’s catch set, the process is listed in that item’s target set. The liveness (or progress) condition—i.e. the circumstances under which a process must execute—would most sensibly be “if a process meets the safety condition (i.e. it can execute), then it must eventually execute”. However, if two processes need the same item, having one process execute will make the other so that it cannot (immediately) execute, so the rule is made slightly more precise and complex: For all processes which meet the safety condition and which need to `catch` the same data item, one of them must eventually execute. In the trivial case, this reduces to the sensible rule stated above. No fairness of access to a particular data item is implied or enforced by the model.

C. Buffering, broadcasting, and multiple readers

Many important optimizations become possible if a process declares one or more of the items within its catch set as *predictable* x , where x is some target set. The declaration is a promise by the process that the item will always be tossed within a finite amount of time after the process begins executing, and that the item will always be given the same target set, i.e. x , when it is `tossed`. Only items with either IN or

NODATA usage can be declared predictable. With this information, a scheduler or runtime system can play some important optimizing tricks. For example, the scheduler itself can `toss` the item when the process begins, since it knows what the target set and data values will be, while suppressing any “real” `toss` for the item from the process. This will instantly make the item available to subsequent processes listed in the target set. To illustrate the power of this optimization, two simple scenarios will be considered.

The first scenario consists of two processes, called Producer and Consumer. Producer has a catch set containing item A with OUT usage, Consumer has a catch set containing item A with IN usage, and Consumer declares A as “predictable {Producer}”. Assume a starting case where item A has a target set of {Producer}. Producer begins running, writes data into item A, tosses A with a new target set of {Consumer}, and finishes. Consumer now starts, and perhaps even before it executes its first instruction, the scheduler tosses a new item A with target set of {Producer}. This allows Producer to run again, starting to put new data into the new item A, even while Consumer is still reading the last version. There are now effectively two copies of item A: that is, item A has effectively been double-buffered. Note that the scheduler might very well opt, in a shared-memory environment, not to prematurely toss A back to the Producer until the Consumer has finished with the item to avoid allocating a new item buffer, as long as the liveness condition is met. Either implementation is valid, since it follows the stated semantics of the model.

The second scenario consists of four processes: Producer, Reader1, Reader2, and Reader3. Producer has a catch set containing item A with OUT usage, and the other three processes have a catch set containing item A with IN usage. Reader1 declares A as “predictable {Reader2}”, and Reader2 declares A as “predictable {Reader3}”. Again, assume a starting case where item A has target set of {Producer}, and again Producer runs, writes to A, and tosses A with a new target set of {Reader1}. Now, the moment Reader1 begins, the scheduler can toss A to Reader2, and when it begins, the scheduler can toss A to Reader3. Three processes can be reading the same version of A. If the three readers are on different distributed-memory processors, this is akin to (and perhaps implemented as) a broadcast. If the three readers are on a shared memory machine, this is akin to three processes with read locks sharing the same memory, all while preserving the serializability of transactions.

D. Functional specifications

Processes have been described as performing transactions, as starting and completing, and as acting very much like subroutine executions. In many languages, it is possible for subroutines to carry internal state from one invocation to the next—e.g. using `static` variables in C, or `SAVE` variables or `COMMON` blocks in Fortran. We forbid it in this model, first

because it is simply not needed (since all of the internal state can be treated as yet another data item which can be `tossed` back to the process at the end of each iteration), and second, because this hidden state can complicate analysis (since it can make each process execution dependent upon the entire history of the process instead of just on information acquired during the current execution). From a practical standpoint, lack of internal state also greatly eases process migration between executions. It should be noted that data like file pointers and random number seeds are considered state, and can only be carried between executions as other state must be—i.e. via a data item.

This leaves the question of input and output. Although processes could conceivably be allowed to use all standard traditional constructs for file and device I/O, this could allow processes to interact with each other (on purpose or accidentally) while bypassing the constructs developed so far, thereby nullifying the advantages of the model developed here. A more clean notion is to treat files just as other data items are treated—i.e. to be caught before using, and tossed before others can use them. For external devices, we will later assume that the devices themselves are capable of `catching` and `tossing` data items, so processes can interact with them indirectly through data items, but other solutions are admittedly possible.

Based on the above and the assumption that processes are expressed in deterministic languages, once a process begins executing, its behavior can depend only upon the values of the data items it `catches` with IN or INOUT usage. Furthermore, the only behavior it can exhibit that can ultimately affect the possible behavior of other processes is the new values it writes to its data items (i.e. the ones with OUT and INOUT usage) before `tossing` them, and the new target set which it assigns to each item as it is `tossed`. In other words, a program in this model is completely specified by giving the set of items (i.e. their names, initial values, and initial target sets), and the following specification for each process:

- (1) a catch set containing item names
- (2) a usage (i.e. IN, OUT, INOUT, or NODATA) for each item in the catch set
- (3) “predictable” declarations for items in the catch set to which they apply
- (4) a description of the resulting target set for each item and the resulting data values for OUT, and INOUT items, as a function of the values found on the IN and INOUT items.

As long as this specification is met, the programming language used to implement the processes, the order or timing in which the processes `toss` their items, and the kind of architecture on which the processes run, are unimportant. A process which enters an infinite loop (“diverges”) before it `tosses` one or more of its items can be specified just as one which does `toss` those items, but with an empty target set, since in either case, no other process may catch the items to

access their contents. (Due to the halting problem, it may not be possible to prove that the specification of a process and the partial function implemented by a programming language are equivalent, but that does not hinder the validity of the approach.)

IV. F-Nets

The description above focused on individual processes, but by its conclusion, each process had been reduced to a fairly concise specification. This section will now describe the model much more formally and abstractly, by treating each process as being equivalent to its simplified specification, with the aim of describing the interactions between these processes. There are, in fact, cases where a single process in the above description must be modeled with more than one of its counterparts (called transitions) in the description to come, but it is a fairly straightforward exercise to recognize the equivalences. The formal description helps to illustrate the kinship between this model and Actors[1], Petri Nets[12], Turing Machines, Finite State Machines, dataflow models, Unity[3], and CCS[11]. The model was originally developed as a refinement of the Large-Grain Data Flow (LGDF) parallel programming approach[2], and early versions went by the name LGDF2[7]. The formal description here is somewhat simplified from its original incarnation[6] primarily because some of the elements of the original description have been passed off to the language level, to be described in a later section.

The model will be described in three parts: its syntax, an operational semantics, and an axiomatic semantics. Due to the different context, the terminology changes rather abruptly here from the previous section, with the following approximate analogs: processes become transitions, data items become frames, a target set becomes a control state, and the names and usages in a catch set become heads.

A. Syntax

An F-Net can be considered as being similar to a Turing Machine, having a (possibly-infinite) tape, separated into *frames*. Each frame contains a mutable data value, called the *data state* of the frame, and a mutable color, called its *control state*. The set of all possible data states is called Σ , and the number of possible colors (other than white) is called p .

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 frame—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 frame. In figure 1, the transitions are shown as circles, the heads as lines, the frames as squares, 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).

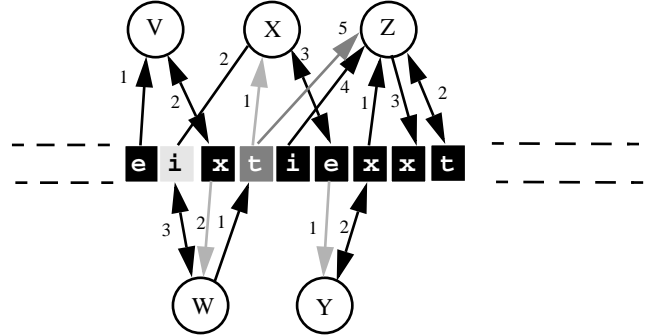


Fig. 1. Sample F-Net in the Turing Machine style

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 $r^{|\Sigma|}$ entries, indexed by the possible combinations of r symbols under those heads. Each entry contains n colored (possibly 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 $\Sigma = \{e, x, i, t\}$.

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

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”.

More formally, an F-Net can be described in set notation. An F-Net Ω of order $p \in \mathbb{N}$ is a 3-tuple $\langle \Sigma, F, T \rangle$ where Σ is the Data Alphabet, F is the set of frames, and T is the set of

transitions. (The colors are represented by the integers 0 through p where 0 represents white and 1 represents green.) Each transition $t \in T$ is a 7-tuple $\langle a, R, W, \kappa, \phi, \beta, \gamma \rangle$ where

$a \in N$ is the number of heads (or “arity”),

$R \subseteq [1, a]$ is the set of readable heads (i.e. having read or read-write usage),

$W \subseteq [1, a]$ is the set of writable heads (i.e. having write or read-write usage),

$\kappa \in [1, a] \rightarrow [0, p]$ is the frame color for predictable heads,

$\phi \in \Sigma^{|R|} \rightarrow \Sigma^{|W|} \times [0, p]^a$ is the firing function,

$\beta \in [1, a] \rightarrow [1, p]$ is the binding of each head,

$\gamma \in [1, a] \rightarrow [1, p]$ is the color of each head.

B. Operational Semantics

An F-Net works as follows. The machine begins in an initial state consisting of a predetermined symbol ι_f and the color green on each frame f . Then, repeatedly, a *ready* transition is located (subject to the *liveness* 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 frame 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 frames 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 frame is also changed to the corresponding symbol from the table entry. (Note that for *read* and *nodata* heads, the symbols in the table serve no purpose other than as placeholders for the color at those positions, and that for predictable heads, even the colors in the table are superfluous.)

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

This semantics suggests that only one transition is evaluated at a time, and if transition evaluation is modeled as taking no time (as they are here), this is sufficient. However, when transitions are implemented as processes as they were in an earlier section, then they may not only take time, but it may not be possible to determine how much longer they will take. That is, the semantics above would require that a scheduler know when a process was finished executing, or more exactly, when the process was not going to perform any more *tosses*, so that the next ready process could be initiated. This means that, to deal with such a process that was computing, a correct sequential scheduler would be required to either solve the (impossible to solve) halting problem, or conversely, to conceivably execute forever waiting for it to finish and thus contradict the required liveness properties.

Fortunately, the operational semantics above 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 frames 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. Changing all of the transition’s frames to white is effectively identical to *catching* items, as described earlier, and so preserves the two-phase (and therefore atomic) nature of the transition evaluations.

There is no need to consider the relative position of frames on the tape, so F-Nets are often represented graphically with the frames 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 frame (rectangle). Read and *nodata* heads with only one colored dot are implied to be predictable. In a black and white medium without shading, 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.

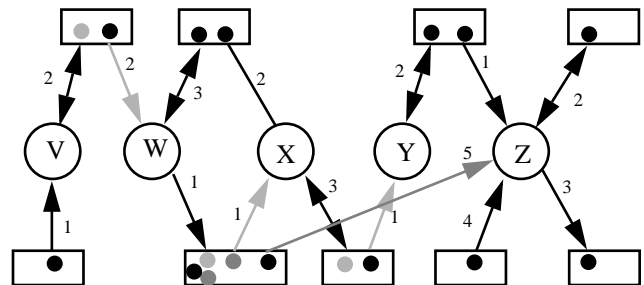


Fig. 2. Same Sample F-Net in Standard Form

C. Axiomatic Semantics

Instead of considering an F-Net as an operational machine-like entity, it can be considered as a means of describing a set of computations—i.e. those which can result from its execution. The specification of the semantics then becomes the characterization of a relationship between an F-Net and the elements of that set of computations. Here, this characterization will take the form of a set of axioms.

An F-Net computation is effectively a trace of the important states and events which occur when following the operational semantics in the previous section, along with the causality between these states and events. A computation takes the form of a directed bipartite graph which consists of state nodes (represented by squares) and event nodes (represented by circles) connected by directed numbered edges (lines). Each event node corresponds to a transition in the F-Net, and represents a firing of that transition during the com-

putation. Each state node corresponds to a tape frame in the F-Net, and represents a state of that frame at some point during the computation. Each state node is labeled with a control state (i.e. color) and a data state (from Σ). The five axioms which relate the computation and the F-Net follow, each accompanied with a description of how the axiom relates back to the operational semantics:

Initial Conditions: Each tape frame f 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 frame f can be reached by following directed arcs. This node will have a control state labeling of green and a data state labeling of ι_f .

Operational explanation: Each frame starts with a control state of green and a data state dictated by ι .

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

Operational explanation: Each frame is accessed by at most one transition at a time.

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 frame under head i of transition t .

Operational explanation: Each transition evaluation can affect all of the frames under its heads.

- (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 .

Operational explanation: A transition can evaluate only when all of its heads are attached to frames of the same color.

- (c) If head i of transition t is a `nodata` or `read` head, then the state nodes attached to edges numbered i will have identical data state labellings. 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.

Operational explanation: `nodata` and `read` heads do not affect the data state of their frames. The frames under all heads are assigned a control state dictated by the firing function, and all frames under writable heads are assigned a data state from the firing function.

Liveness: If there is some transition t and some set of state

nodes corresponding to the tape frames under 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 frame, then at least one of those state nodes must have an outgoing edge.

Operational explanation: A transition will remain steadily ready for at most a finite amount of time before evaluating.

Time: The graph will be acyclic.

Operational explanation: Causality moves with time, which moves in only one direction.

See figure 3 for an example of a partial execution graph for the F-Net in figure 2. Rather than label each state node with the appropriate frame, we here organize all state nodes referring to the same frame to be in the same horizontal line.

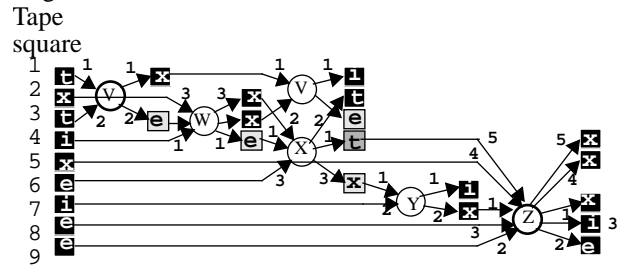


Fig. 3. Sample execution graph (partial)

The axioms can be defined more formally by expressing the basic form of a computation in set notation, and rewriting the axioms to relate the mathematical entities of the F-Net and the computation. (Readers not requiring this rigor can safely skip to the next section.)

Def: A computation $\chi_{\Omega, \iota}$ for F-Net $\Omega = \langle \Sigma, F, T \rangle$ with initial marking $\iota: (F \rightarrow \Sigma)$ is a 6-tuple $\langle E, S, \circ E, E^\circ, \tau, \phi \rangle$ where

E is a set of events

S is a set of states

$\circ E \subseteq S \times E$ is the set of “before” edges,

$E^\circ \subseteq E \times S$ is the set of “after” edges,

$\tau: (E \rightarrow T)$ is an transition event labeling,

$\phi: (S \rightarrow F)$ is a frame state labeling,

such that functions

$h: E^\circ \cup \circ E \rightarrow N$, called a head labeling

$\dot{\phi}: S \rightarrow [0, p]$, called a control state labeling,

$\hat{\phi}: S \rightarrow \Sigma$, called a data state labeling,

exist, and axioms 1 through 10, described below, hold.

The axioms will be based on the following notation and definitions.

Notation: $\phi(s)$, $\dot{\phi}(s)$ and $\hat{\phi}(s)$ are usually written as \bar{s} , \dot{s} and \hat{s} , respectively. Implication, often represented $a \Rightarrow b$, is here represented $\frac{a}{b}$. Fixed sequences, like tuples, are shown in angle brackets, and a single subscripted element in angle brackets represents a sequence showing a representative element. Adding a vector symbol over a set of unique integers (e.g. \vec{W}) creates an ordered sequence from the set, and a function applied to a sequence yields a sequence con-

sisting of the function applied to each element—i.e. $f(\langle x_1, x_2, \dots, x_n \rangle) = \langle f(x_1), f(x_2), \dots, f(x_n) \rangle$. Element x of tuple y is represented x_y .

Def: The relation $x \ll y$ between two nodes of a computation (i.e. $x, y \in E \cup S$), pronounced “ x **precedes** y ”, is defined to hold if and only if x and y are the same node, or you can get from x to y by following edges—i.e. $x \ll y$ iff

$$(x = y) \vee (\langle x, z \rangle \in E^\circ \cup {}^\circ E) \wedge (z \ll y)$$

Def: The notations n° and ${}^\circ n$, called the **successor** and **predecessor** sets of node n , respectively, are the nodes just after and just before n in the computation—i.e.

$${}^\circ n \text{ is defined as } \{x \mid \langle x, n \rangle \in E^\circ \cup {}^\circ E\}$$

$$n^\circ \text{ is defined as } \{y \mid \langle n, y \rangle \in E^\circ \cup {}^\circ E\}$$

Def: The set \underline{S} , called the **initial states**, denotes the set of state nodes in the computation (if any) which precede all other state nodes having the same name — i.e.

$$(\underline{s} \in \underline{S} \wedge s \in S \wedge \bar{s} = \underline{s}) \Rightarrow \underline{s} \ll s$$

Def: The notation $X \xleftrightarrow{\alpha} Y$, pronounced “ α **bijectively maps** X **to** Y ”, is defined to hold if and only if α maps the set X onto the set Y — i.e. iff

$$(\forall y \in Y \exists x \in X \mid \alpha(x) = y) \wedge (\forall x, x' \in X \mid (\alpha(x) = \alpha(x')) \Rightarrow (x = x'))$$

The five axioms which define the valid computations $\chi_{\Omega, \iota}$ producible by F-Net Ω :

1) Initial conditions

Each frame in F has an associated initial state in S with a control state of green (i.e. 1) and a data state equal to the initial state of the frame— i.e.

$$\frac{f \in F}{\exists \bar{s} \in \underline{S} \mid \bar{s} = f \wedge \dot{s} = 1 \wedge \tilde{s} = \iota(f)}$$

2) Atomicity

Each state node has at most one incoming edge and one outgoing edge — i.e.

$$\frac{s \in S}{|s^\circ| \leq 1 \wedge |{}^\circ s| \leq 1}$$

3) Firing

a. The incoming and outgoing edges of each event node will be numbered for the heads associated with the transition corresponding to that event—i.e.

$${}^\circ e \xleftrightarrow{h} [1, a_{\tau(e)}] \wedge e^\circ \xleftrightarrow{h} [1, a_{\tau(e)}]$$

c. Each of the outgoing edges will be attached to a state node representing the same tape frame as that under the corresponding head—i.e.

$$\frac{\langle e, s \rangle \in E^\circ}{\beta_{\tau(e)}(h(\langle e, s \rangle)) = \bar{s}}$$

d. Each of the incoming edges will be attached to a state

node representing the same tape frame as that under the corresponding head, and having the same color as the head—i.e.

$$\frac{\langle e, s \rangle \in {}^\circ E}{\beta_{\tau(e)}(h(\langle e, s \rangle)) = \bar{s} \wedge \gamma_{\tau(e)}(h(\langle e, s \rangle)) = \dot{s}}$$

e. The data states of the writable result state nodes and the control states of all the result state nodes are dictated completely by the data of the incoming state nodes and the firing function of the transition associated with the event node—i.e.

$$\frac{\langle \bar{r}_j \rangle = \beta_t(\vec{R}) \wedge \langle \bar{w}_j \rangle = \beta_t(\vec{W}) \wedge \langle \bar{s}_j \rangle = \beta_t(\vec{[1, a_t]}) \wedge \langle r_j, e \rangle \in {}^\circ E \wedge \langle e, w_j \rangle \in E^\circ \wedge \langle e, s_j \rangle \in E^\circ}{\Phi_{\tau(e)}(\vec{r}_1, \vec{r}_2, \dots) = \langle \tilde{w}_1, \tilde{w}_2, \dots, \dot{s}_1, \dot{s}_2, \dots, \dot{s}_t \rangle}$$

f. The data state under heads having no write permission will not be altered by this event—i.e.

$$\frac{\langle e, s \rangle \in E^\circ \wedge \langle s', e \rangle \in {}^\circ E \wedge h(\langle e, s \rangle) = h(\langle s', e \rangle) \notin W}{\tilde{s} = \tilde{s}'}$$

g. The control state for predictable heads will be as predicted—i.e.

$$\frac{\langle e, s \rangle \in E^\circ \wedge \kappa(h(\langle e, s \rangle)) \neq 0}{\dot{s} = \kappa(h(\langle e, s \rangle))}$$

4) Time

The computation graph is acyclic—i.e.

$$\frac{x \ll y \wedge y \ll x}{x = y}$$

5) Liveness

If there is a set of state nodes after which an event node could be added, then at least one of those state nodes will have an outgoing edge—i.e.

$$\frac{(\bar{s}_j) = \beta_t(\vec{[1, a]}) \wedge (\dot{s}_j) = \gamma_t(\vec{[1, a]})}{\exists j \mid |s_j| \neq 0}$$

V. Observations and Implementations

This section contains various insights and comments about the F-Nets model. Though formal proofs will not be provided here, proofs of some of these properties can be found in related work[6]. First, some of the properties of the model relating to non-determinism are covered, followed by its relation to other computational models, then implementation strategies are briefly mentioned, and finally some observations on the relationship between F-Nets and structured programming are explored.

A. Non-determinism

By removing the state nodes from a computation such as that in Figure 3 (i.e. by merging the incoming and outgoing edges of each square node into a single edge), the computation becomes a partial ordering (directed acyclic graph) of

event nodes, where each node represents the evaluation of a (firing) function. Looked at in this way, the computation as a whole can be viewed as the composition of functions, implying that the computation is itself represents a function, being evaluated on the initial state represented by ι , which can be considered its argument. Re-inserting the state nodes shows that they can be regarded as data being passed from one function evaluation to the next, implying that the data and control state (i.e. value and color) of each state node is uniquely determined by the structure of the computation and ι . Thus, the functions $\dot{\phi}$ and ϕ in the last section are uniquely defined for a computation. Incidentally, h is also unique, since the head represented by each arc can always be determined by the frame and transition labellings of its end nodes. This explains why these three functions are not part of the computation itself.

However, the computation itself is not necessarily uniquely determined by the F-Net and the initial state. Such nondeterminism can be a desirable characteristic as long as it is not introduced by accident—for example, to make the computation flexible enough to take timing differences into account to create “first-come first-served” behavior. Potential non-determinism in an F-Net can be detected syntactically, so tools can allow the programmer to verify that it is desired before the program is run. 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 frame (say $s1$) and those same transitions do not have differing-color heads on another “shared” frame (say $s2$). This is easy to see: If each frame has at most one head with each color attached to it, then the color of the frame will always uniquely determine the next transition that can evaluate using that frame, so the event node following each state node in the computation will be uniquely determined, thereby uniquely determining the computation as a whole. Even if some frames have multiple heads of the same color, as long as some other frame dictates the order in which the corresponding transitions must evaluate, the reasoning still holds.

Note that the above lists the conditions under which an F-Net “may be” non-deterministic. Such F-Nets may also be deterministic, but this may not be syntactically apparent. For example, even if multiple transitions share a single frame with like-colored heads, they may each be attached to other disjoint frames with other heads, and the internal workings of the F-Net as a whole may conspire to ensure that these other frames allow only one of those transitions to be ready at a time. Also, different computations can represent identical functions, due to commutativity, associativity, or other syntactically unapparent equivalences.

Because non-determinism is always associated with like-colored heads on a shared frame, it is easy and efficient for tools to trace it and to replay these programs with minimal overhead. Specifically, by attaching a log onto each frame having like-colored heads, and recording the order in which

the associated transitions access those frames (which can take as little as one bit per evaluation of those transitions, and no bits for other transitions), the computation can be reconstructed at a later time with the logs and ι .

B. Relationship to Other Models

The description of the F-Net model has been based loosely on the Turing Machine. Likewise, it can be interpreted as a Turing Machine coordination model. That is, each transition could be considered a multi-tape TM, and (given some way for a TM to signify the tossing of a tape) each F-Net frame could be considered as a TM tape being passed from one TM to another.

A kinship between F-Nets and Petri Nets is suggested by the similar appearance (though the notations are confused). Specifically, an order p F-Net (i.e. one whose frames can collectively assume up to p colors) 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 frame to p Petri-Net places. 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 frame. 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 Σ 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 frames in an F-Net represent updatable containers more like variables in traditional imperative languages. Also, F-Nets more naturally accommodate non-determinism.

Each frame 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 much theory could carry over from CCS to F-Nets.

By restricting F-Nets to use only green control states, a

model similar to Unity[3] emerges, although the latter has some additional fairness constraints and uses guards to take the place of control states.

C. Real-World Implementation

Because F-Net transitions can be implemented in standard programming languages (augmented by some special constructs to handle “`toss`” statements), the primary implementation questions come in how to efficiently schedule transitions and migrate frames—in other words, dealing with control and data state.

Rather than maintaining control state separately for each transition, it is often more efficient to simply reflect their presence using a kind of reference count associated with each transition. That is, each transition holds a count giving the number of its heads which are attached to a frame of the wrong color, and each time a transition is initiated or executes a `toss` statement, the appropriate reasons counts for related transitions are adjusted (within a critical section), and new transitions are scheduled (i.e. initiated) whenever their reasons counts reach zero. In a shared-memory environment, the granularity of critical sections can be reduced (i.e. their number increased) by static analysis of the F-Net’s topology[8].

In distributed memory environments, `toss` statements associated with `write` or `read-write` heads often map straightforwardly into message `sends` which pass the data state associated with the frame to the next process that will read it. There are, however, some circumstances where the next process to read the data is not immediately known. In that case, the data state can either be left with the `tossing` 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. Cooperative Data Sharing (CDS) is a simple subroutine package which embodies the communication requirements of F-Nets without the dataflow-like run-time semantics[5]. Although a major goal of CDS was to aid in the implementation of F-Nets on real computers, it can also be used as a standard communication substrate for a variety of other purposes. In message-passing environments, it serves a role much like MPI and PVM, but it also works efficiently in shared-memory environments because of the lack of copying allowed by its semantics[4]. Like F-Nets, CDS employs a logical model which may be expressed in different physical implementations.

Using the above techniques, `tossing` a frame to another transition on the same processor has almost negligible cost—it often requires only that a reasons count be decremented, and that the other process be initiated (akin to a subroutine call by the scheduler). Often, no data needs to be copied, since the frame data can be accessed in place by both the `tossing` and `catching` transition. This has the important result that the granularity of a program can be dynamically increased at execution time, so one can theoretically write a program once and

run it efficiently on different numbers of processors.

Ensuring that a particular scheduler and runtime system actually implement the axioms is itself a somewhat theoretical problem. This can be performed by defining a partial ordering over the set of all possible computations (i.e. a partial ordering of partially-ordered sets) based upon a computation being more well defined when it has more nodes or when those nodes are more well defined, then showing that the runtime system monotonically advances the computations in this p.o., and that the fixpoints satisfy the axioms[6].

D. On algorithms, computations, structured programming

Definitions for the term “algorithm” are rare, but Knuth (for example) once defined one as a set of rules which gives a sequence of operations for solving a specific kind of problem[10]. Using common vernacular, we will call such a “sequence of operations” a sequential computation. As already mentioned, this definition of algorithm is somewhat outdated, and inapplicable to parallel environments, but it elicits another question: What kind of representations do we use for such algorithms, in order to represent sequential computations? Back when Knuth originally expressed this definition, his algorithms consisted of simple steps, each representing some mapping of state, built into short sequences, and punctuated by conditional and unconditional branch statements. Programs of the time looked the same. To compute, one followed the steps, and the computation (sequence) unraveled from it like a piece of spaghetti, leading some to eventually call it spaghetti coding. Working backward, such algorithms can be considered as raveled-up computations. That is, instead of representing the entire computational sequence, the algorithms saved space and became more general by folding up the algorithm and unfolding it as it was executed.

More recently, spaghetti coding has given way to structured programming, a more organized method of expression that typically uses indentation to increase the dimensionality of the program slightly to visually provide another measure of closeness between two steps. In addition to the physical closeness of two steps in a sequence, there is also the implied closeness of steps terminating indented blocks (such as those at the beginning and end of loops or conditional branches). As a result, when a computation unfolds or unravels from the algorithm, operations which are close in the computation are also visually close in the algorithm. It allows the algorithm to more visually represent the computation.

In parallel programming, parallel computations have long been recognized not as sequences, but as partial orderings of operations[13], and partial orderings are representable by directed acyclic graphs, or DAGs. In traditional parallel programming, the correspondence between the “parallel algorithm” (expressed as a set of communicating sequential algorithms) and the resulting parallel computation(s) is often

not apparent. (Also, because it is so much easier to express ordered than non-ordered operations when using this method of programming, it often coerces programmers to order logically-unrelated operations, even when it is not to the advantage of the programmer nor the computer.)

Computations in the F-Net model are DAGS, or partial orderings of operations, and if viewed correctly, they exactly represent an unfolding of the associated F-Net. For example, if one were to take the computation in Figure 3 and fold each event node so that the incoming state nodes fell exactly on top of the outgoing state nodes, one would eventually end up with a picture similar (and topologically identical) to the F-Net shown in Figure 1. Such folding is familiar to Petri Net analysts. We believe that these properties give strong argument to considering F-Nets as structured parallel algorithms.

VI. 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 deficiencies for programming.

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.

A. 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 is meant to be analogous to a custom CPU chip, and is the only construct in SC which transforms data. In the real world, it is a program (or subroutine) like those discussed in section III, and its body is constructed with tools outside the realm of SC, but SC logically regards each chip as acting in a specific deterministic way when initiated: The chip initializes all of its internal data, then reads one data item from each of its read or read-write pins, then computes for some amount of time and writes one data item to each of its write and read-write pins. In addition, for each of its pins, it posts 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, the 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, data type, and signal set. The permissions of the receptacle 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 which is 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 overridden 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 special repository associated with the program, called the *program data base* (pdb).

Wires are represented in the board diagram as colored lines, and they connect each of the receptacles of each 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*, on the memory end for *write*, on both ends for *read-write*, and on 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 frames, 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.

B. 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 null (i.e. 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 there is a matching signal color, the *i-memory* becomes colorless once again, and the

matching signal is effectively posted through the target wire to the target memory.

C. 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.

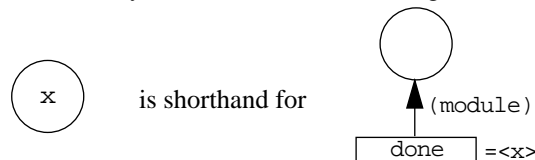


Fig. 4. Labeled circle shorthand

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. First, the socket reads the descriptor on its module receptacle, but 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. Second, even though the `(module)` receptacle looks predictable, it is so only if the module thereon is a chip. If the module is a board, the `done` signal is never posted to the receptacle unless explicitly directed, as described next.

A board is permitted to have an *i-memory* named `(module)`, with one signal named `done` in its posting table. This *i-memory* naturally corresponds to the `(module)` receptacle

on the board, and changing its color to the color of done in its posting table will have the natural effect of causing the socket containing the board module to post a done 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 possess color at that time, they will continue to possess 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.

Simply put, posting the done signal to its (module) receptacle allows a board to finish up what it started before disappearing, but does not allow it to start more work. The board can be (and is) considered to have logically finished as soon as it has posted this signal. In atomic transaction terminology, this ensures that the board is entering a shrinking phase. If the programmer ensures that the board does not enter a second growing phase 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 (1) it can copy any kind of data object, including module descriptors, and (2) it can do so very efficiently (sometimes without even performing an extra copy) because the SC scheduler knows the desired result and can therefore optimize.

D. 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 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 to one instance through one board is cumbersome to say the least. 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 module from its object pin, and assuming that it is a board with a non-zero instantiation level, 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 the object 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 so-called “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 updates 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. “class variables”, “superclass variables”, etc.).

E. 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. (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.

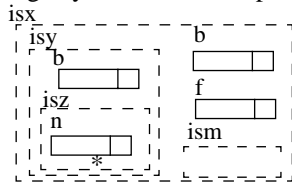


Fig. 5. I-sets. Roughly, $isx = \{isy = \{b, isz = \{n, *\}\}, b, f, ism = ?\}$

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 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.

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.

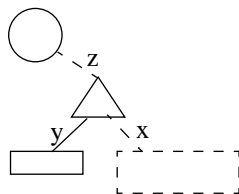


Fig. 6. Bundling wire y and cable x into cable z

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.

F. Arrays

Memories within SC, like tape frames 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.

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 on 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.

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.

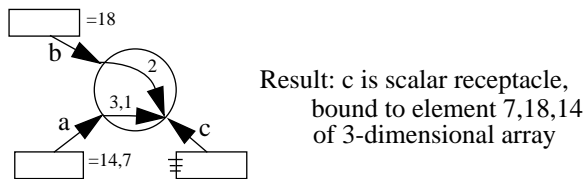


Fig. 7. Two selection bindings: scalar from 3 dimensions

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 only 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 effectively organized into an array shaped just like the primary receptacle. The dimensionality of the result is therefore effective-

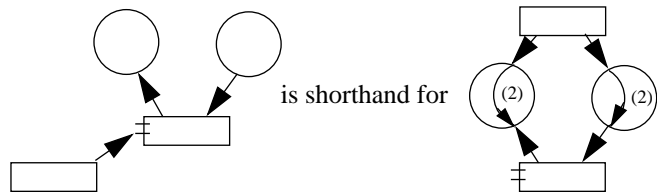


Fig. 8. Shorthand for range selections (like dimensioning) effectively increased by the dimensionality of the primary receptacle, just as it was decreased by the length of the index list. (Ranges are not allowed with non-scalar primary receptacles.)

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 in conjunction with a selection representing a stencil.

G. 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.

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 its 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.

VII. Conclusion

F-Nets show significant promise for enabling the scalable design and construction of portable structured parallel programs, and Software Cabling illustrates how this model can

be used in conjunction with current programming technology to facilitate programming. Techniques such as these are becoming extremely important in light of the recent trend toward computational grids—i.e. widely distributed clusters of powerful heterogeneous machines, shared by collections of agencies and institutions to handle extremely high computational loads in an efficient and cost-effective manner. However, the F-Net approach must still be further validated on real workloads, and that will require the further development of programming tools and runtime systems, based at least partially on the research presented here.

ACKNOWLEDGMENTS

The F-Nets model was developed as Ph.D. dissertation work at the Oregon Graduate Institute, where it began with the research and helpful feedback of Robert Babb (thesis advisor), and was furthered by feedback from the other dissertation committee members—Michael Wolfe, Harry Jordan, and Richard Kieburtz. Early discussions with Dick Hamlet also proved insightful. The CDS package was developed at NASA Ames Research Center, where support was provided by the NAS and HPCC programs. Doreen Cheng provided significant feedback in an early documentation effort for SC.

REFERENCES

- [1] G. Agha, “Actors: A model of concurrent computation in distributed systems”, MIT Press, 1986.
- [2] R. G. Babb II 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.
- [3] K. M. Chandy and J. Misra, “Parallel Program Design: A Foundation”, Reading, MA, Addison-Wesley, 1988.
- [4] 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.
- [5] D. C. DiNucci, “CDS”, <http://www.nas.nasa.gov/Tools/CDS>
- [6] D. C. DiNucci, “A formal model for architecture-independent parallel software engineering”, Ph.D. Dissertation, Oregon Graduate Institute, 1991, also available at <ftp://cse.ogi.edu/pub/tech-reports/1991/91-TH-007.ps.gz>
- [7] D. C. DiNucci and R. G. Babb II, “Design and implementation of parallel programs with LGDF2”, COMP-CON’89, San Francisco, 1989, pp. 102-107.
- [8] D. C. DiNucci and R. G. Babb II, “Practical support for parallel programming”, Proc. 21st HICSS Software Track, 1988, IEEE, 109-118.
- [9] 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.
- [10] D. E. Knuth, “The Art of Computer Programming: Volume 1/Fundamental Algorithms”, Addison-Wesley, Reading, MA (1975).
- [11] R. Milner, “A calculus of communicating systems”, Lecture Notes of Computer Science, vol. 92, Berlin, Springer-Verlag, 1980.
- [12] J. Peterson, “Petri net theory and the modeling of systems”, Englewood Cliffs, NJ, Prentice-Hall, 1981.
- [13] V. R. Pratt, “Modeling concurrency with partial orders”, International Journal of Parallel Programming, vol. 15, 1(February 1986), pp. 33-71.