Temporospatial Software Architecture With F-Nets and ScalPL

David C. DiNucci
dave@elepar.com
October 2017

Abstract
Although traditional sequential computing is ubiquitous, and designed to be natural for humans to grasp, its zero-dimensional nature fails to recognize or exploit the natural potential of the 3-dimensional space in which it exists. Still, in spite of the constraints and inefficiencies imposed by these contrived sequential approaches, we hold tight to them due to the difficulty of top-down design and loss of simple functional specifications for modules when we introduce concurrency. Even when moving in the direction of concurrency/parallelism, we tend to cling to sequential approaches with minimal extensions, only exacerbating the problems. Dataflow models (especially large/coarse grain) help to address some of these deficiencies, but only by introducing some other complexities and inefficiencies. The F-Nets formal computational model helps to retain dataflow advantages without many of the costs. ScalPL (Scalable Planning Language) builds upon F-Nets to provide modern real-world software constructs and design methodologies while retaining its scalability, portability, efficiency, and language independence.

Problem statement and background

Programming as we have known it is rapidly becoming obsolete — or more precisely, inadequate. This shouldn't be surprising: Ever since computers were first developed in the '40s, they've been largely based on a Turing or Von Neumann model which bears little resemblance to any natural physics or geometry. This artificial computing environment was contrived from, and simulates, the way a (single) human might naturally think about solving a problem, recording that solution, and then carrying out that solution (algorithm or plan). This means that all computation is performed at a single point in space by one entity ("brain" or CPU), and thus as a sequence of operations, and therefore all information needed or produced by the computation must be moved to and/or from that point in space. The overhead (in energy, time, and materials) to facilitate this very contrived world becomes enormous as we have pushed it to meet our more advanced needs.

And the preservation of this approach has achieved little. Such a simplified architecture served a purpose as we were just getting our feet wet writing programs, and at speeds where physics didn't interfere too badly. When computer architects eventually found the sequential nature obstructing performance, especially regarding the time to get data from memory, they built instruction-level parallelism and out-of-order execution into the processors, doing their best to hide concurrency inside the chip while simulating a sequential model to the programmer, seemingly for their benefit in simplicity. But then, abruptly, other physical issues intervened. Since about 2000, it became clear that even with such tricks, we couldn't make processors much faster, due largely to energy/heat factors. While component costs and sizes continued to shrink, we have been pushed in the direction of multiple processors, like it or not —as multiple processor cores on a single chip, or on separate chips.

But there are more reasons for this evolution away from point-based computing than just heat

dissipation. The cloud, big data, and “bandwidth everywhere” have geographically distributed the location of both the data and the execution of multitudes of programs, often making it hard to discern where one program ends and another begins. A complete program is now seldom conceived, or recorded, or executed in a single point in space. It is often better for some parts of the program to execute closer to the data than to the ultimate user, and other parts vice versa, and those decisions cannot necessarily be made up front, before we know the distances and latencies involved. And those factors together with globalization of technology and open source have made almost every program a collaboration. So the question is no longer whether we can or should evolve toward a concurrent model, we know we will, we just need to know the best way to do so.

To now, we’ve been going there, but only kicking and screaming, still with the notion that the more we can keep our thinking and coding in the sequential realm with sequential languages, and out of concurrency and interaction, the safer we’ll be. So even though some technical fields have been exploring parallelism and distributed systems for decades for performance, the focus has remained on the sequential processes and threads—i.e. anchored computing points—and then how to sequence operations at those points to get data back and forth between them when and where needed. In other words, with few exceptions, the interactions themselves between the pieces are never explicitly and independently specified or shown, but are instead emergent behavior, the result of hopefully mutually productive operations in the processes at the endpoints occurring at the right times. For massively parallel applications, we have come up with a few techniques (SPMD, collective communication, parallel loops) by which we can at least encode an entire interaction in just one place which is then replicated as necessary, but these are at best special cases. If differing processes, perhaps implemented by differing groups, must interact, very tricky coordination is required.

It is time to make the full evolution, to where those stable points of processing are not considered “the program” any more than the instruction set of a computer is considered a computer program. A program consists of the interconnections and sequencing between its basic parts, in the way it transforms data over time, in the overall mapping from inputs to outputs. Just as sequential programming has dealt with these factors directly, and developed tools and techniques for manipulating and visualizing them, so should concurrent programming. The old adage is that “the network is the computer”. We must now accept that “the interactions and transformations are the program,” with those that take place over longer distances garnering as much respect and care as the others within a single process. In fact, it may not even be possible to know what distances or number of processors will be involved until runtime, so any concurrent program should execute efficiently on smaller numbers of processors, or even one: They must effectively have variable granularity.

Exascale

One big push the last few years, to test our ability to harness massive parallelism, is toward exascale computing. The motivating question there is simple: What do we need to do to achieve one exaflops (1 quintillion, or 10^18 floating point operations per second) for a single application? The answer is not at all simple, and may not even be possible, depending on the definitions used. Getting enough parallelism is one hurdle, but even then, heat remains a major one: Simply moving too much data or data too far on a single chip, much less between chips, can be the limiting factor. Extra data movement can also come from too much copying for communication (such as to simplify programming by simulating that the originator and receiver are far apart even when they’re not), or in recording the state of the program (“checkpoints”) to recover from the faults or failures which become almost a certainty when very large numbers of processors are executing for long periods of time.

Regardless of our success in reaching exaflops (or the practicality of that goal), it confronts us

with the same kinds of issues discussed above: Making the program about its data transformations, and letting the best locations and communications for the job and platform be derived and decided at a later time, after the program is completely written (perhaps during runtime), to optimize data movement and thereby performance and energy consumption.

Functional specifications

Our aversion to a concurrent programming model which embraces interaction runs much deeper than just a dearth of tools and languages, to the underlying engineering of systems.

A vital aspect of building any large structure, whether in our physical world or in the ones and zeroes of a computer's storage, is the ability to build large things from well-understood smaller ones, where the rules of assembling the small ones, and the potential interactions between them, are easily fathomable. This includes the ability to break large systems down into smaller components. Specifications play a prime role in this. Instead of smaller components being described only by their instantiation/implementation, and larger ones as the sum of all of those implementation details, smaller ones are better described more abstractly in terms of some behavioral specification, in just enough detail so that any component meeting the spec will function correctly in that place. (In the best of cases, the specs of the larger system can be derived automatically from that of their components and the way they are interconnected.) When working from a high-level system spec, it's convenient to either find existing reusable components with suitable specs that can be used in its implementation, or to create new component specs on the fly, understanding that they will be implementable later. These component specs also come in handy when debugging a system: If a component is found to not meet its spec, that's a likely cause of problems.

While working out the architecture of a large sequential program/system, this approach works fine, to systematically posit the existence of a function, subroutine, method, etc., that will act a certain way, and insert an invocation of that routine. If a routine with those specs doesn't already exist, we now have a partial spec to guide its implementation. Early languages only provided functions or subroutines for this, then object oriented languages made it easier to keep track of (and more abstractly define) state changes, etc., by encapsulating the state and its transformations into a class (instance) with the functions that accessed it. The engineering is rather straightforward, for a few reasons. One is that this routine we are invoking will do its work with the inputs/arguments we give it and then get back to us with results, synchronously, before we continue, so our mind can be in just one place at a time during the design as it is during the execution. Second, the composition of functions is indeed quite well understood, and a function (specifically a partial recursive function) composed with others will always yield yet another partial recursive function: It's functions all the way down. And a third, because of that, a quite adequate specification of the routine is the function it computes — i.e. the mapping from arguments and old program state to results and new program state — without restricting the way that mapping is implemented. (Other specification methods may additionally hide the specific program state by defining the input/output mapping in terms of history of the object to which it refers/belongs.)

Contrast this with building a concurrent system using traditional sequential languages. When handing off some work to a concurrent process, that process will usually not synchronously return a result, and in fact may send any "result" to some completely different process. Even if a result does come back, it may be later, and affected by other inputs from other processes, so the convenience of understanding it all with a single train of thought is long gone. We have control and data flowing all over, and need to keep track of it all. And even though each process can technically be defined as a partial recursive function in terms of its output streams as the result of its input streams, the long life of these processes, and the fact that every subsequent output can depend on the program state which can in turn depend on every

input that came before it, can make such a specification incredibly complex and virtually useless for any one interaction. And even that complex spec would not be adequate to characterize how the process will behave in the context of other processes: One must also include the circumstances under which the process is wiling to accept inputs or produce outputs, such as the order in which different streams can or must be accessed externally for progress to occur. And even with such a specification for each process, understanding the potential results of composing such processes together adds another level of complexity. In fact, the composition of such processes through communication may yield something which is not a function at all, but instead nondeterministic—and it is often not obvious when the resulting composition is or is not functional vs. nondeterministic.

So, as engineers, with every interprocess communication we propose, we are distracted to consider things outside of our original train of thought, perhaps being led far away, to resolve loose ends of data and control flows before returning to the original task at hand, with no very good techniques, tools, or even rules of thumb for managing or visualizing these loose ends. In fact, resolving any one may require resolving some or all the rest, leaving us with a mound of intricate interrelated dependencies. The engineering process becomes unmanageable.

Data parallelism: Non-looping loops

But even if we could tackle those issues, handing occasional tasks off to other processors achieves only limited concurrency, and so limited potential at speeding things up by having work actually happening at the same time, sometimes differentiated as *parallelism*. To have an impact there, we generally seek many orders of magnitude more concurrency, the most common and obvious source being data parallelism, the processing of large data sets where operations to individual elements or groups thereof can occur mostly independently. Even here, process parallelism can still provide small multiples of additional performance. In sequential programming, there's virtually no advantage to considering independence between processing of each element: If the same procedure must occur to multiple items, we have no choice but to do them sequentially, so there's no harm in acknowledging that one will follow the other, and in fact, to take advantage of that fact when helpful, leading many to speak of “loops” even when considering concurrency. But in a concurrent context, we actually want the opposite of a flow of control looping back and repeating: We want to express replication of operations, not loops, even if some of that replication will be expressed serially in processor-constrained environments (or where pipelining can be exploited such as GPU processing).

Traditional focus on parallellsm, especially in the scientific and “big data” communities, has indeed centered on data parallelism. With many of these approaches, the program is either considered as shifting between data parallel modes (phases, or epochs) and sequential ones, so data parallelism is treated as more of a mode that one enters or exits, instead of just one more tool to use when expressing the problem at hand which might lead to ample concurrency. This has led to terms like “fork-join” parallelism, to express that a program forks out to get parallelism, then rejoins into often a single sequential thread to process the results.

Other approaches

Several approaches have been proposed and/or tried over the years to address at least some of these issues. For example, Kahn (or Kahn-McQueen) process network theory [1] demonstrated that by imposing certain restrictions on how processes inter-communicate, the resulting network of processes can be assured to be deterministic, somewhat simplifying analysis. The Actors model [2] simplifies analysis of interactions by breaking large long-lived stateful processes into sets of individual actors which each take messages and, as a result, alter their internal state (future behavior) and/or send messages to other actors. Hoare's CSP

[3] formalized the complex behavior of (potentially long-lived stateful) interacting processes, showing the immensity of the problem of specifications for concurrent systems.The Ptolemy project [4] has similar goals and philosophy to the work that will be presented here, but the approach is significantly different.

Data parallelism has been explored in a number of approaches, from MPI collective communication to Hadoop to NESL to Parallel Fortran constructs.

Large Grain Data Flow offers a path forward

Since text is sequential by nature and not conducive to depicting lots of interacting pieces, our natural intuition pulls us in the direction of wanting to visualize these concurrent processes as a picture somehow, and many approaches have been tried. These have fallen down in several ways. While any program must express both the flow of data and the flow of control, for sequential languages, the tendency is to concentrate on the single control thread (specified by the sequencing of statements and control statements), and establish the flow of data as required by associating variable references there with their previous usages (or definitions). In a concurrent diagram, the several data flows may or may not follow the flows of control, so merging all that info into one diagram is difficult or confusing, and some approaches don't even try, just creating different diagrams for each. But although these multiple diagrams may serve as sort of a map or documentation for the program, they don't provide enough information to be the program itself. And even if we could document how data and control flowed between long-lived processes, the complexity of the processes themselves and their interactions would still obscure the overall behavior.

Dataflow execution models can move us toward a solution. In their most pure form, they consist of individual fine-grain operations (functions) which are connected by directed arcs (arrows), essentially paths over which data can flow. A node, representing a function or operation, can evaluate, or *fire*, when all of the incoming arcs have at least one data item, or *token*, and when the function evaluates, it consumes a data element/token from each incoming arc and puts result tokens on one or more outgoing arcs, thereby potentially enabling other nodes to fire. Although it is not strictly necessary that the operations be stateless, and therefore that the results of each firing deterministically depend only on the tokens on its arcs when it fires, that does simplify analysis and state migration, and if state can be fed back by a loop-around arc anyway, no power is lost in that restriction. By establishing that the evaluation/firing of each operation is independent of any previous firings, and that the mappings from inputs to outputs are functional, the resulting composition is both fairly easy to visualize and is also functional, and may benefit from concurrency whenever multiple nodes are ready to fire at the same time. Since the flows of control naturally follow the flows of data, this can all be encompassed fairly directly in one diagram.[5]

These dataflow diagrams do satisfy many of the goals we outlined at the beginning. They allow the architect to focus on and manipulate the high-level data and control flow of the system in a very direct and obvious way, and the parts of the program are again easily composed functions, which (if composed carefully) result in functions as a result. So, in architecting the program, functional specifications are (again) suitable for the parts, as with a sequential program.

Unfortunately, there are a number of practical and theoretical issues with the approach, dealing primarily with the nature of the arcs. If an arc is considered as a queue, allowing multiple tokens/items to gather there to be consumed as needed, then is there a limit to how many? If not, then the model has a rather extravagant, inefficient, hard-to-implement feature built into every interaction. A limit eases things, but one must still include that limit in any reasoning, and if the limit is not explicit, or if tokens are lost by exceeding the limit, it is made

even worse. Some dataflow systems treat an arc as “single assignment”: Once it is given a value, it maintains that value from then on, no further node firings are allowed which would redefine it, and everything else that relies on it is allowed to fire. That is rather efficient and portable, but not very expressive, as there’s no ability to create cycles or feedback, and the diagram is essentially “one way”, acyclic, essentially a computation instead of a program, where each node can fire at most once. To remedy this, such systems rely on languages that dynamically build more dataflow graph. SISAL [6] is such a language. While that may be workable (and is similar to functional languages), the constructs to create such graph are generally themselves not graphical, and thus don’t meet our needs here.

Simply put, we often don’t need anything as complex as a dataflow arc to manage these inter-node interactions anyway. The arcs are both too powerful, providing functionality that is often not needed (like queuing), and too restrictive, in that they are unidirectional, so aren’t good at facilitating updating shared information, where either or both originator and consumer are not known in advance. Also, the very idea of mutable memory, updating things in place, is rather foreign to this model, even though very natural for computers in generl.

So we ask: Can we get some of the advantages of this high-level dataflow style of programming without the drawbacks, by replacing the data flow arcs with something else?

F-Nets: Graphical dataflow with no flow

The simple answer is: Yes

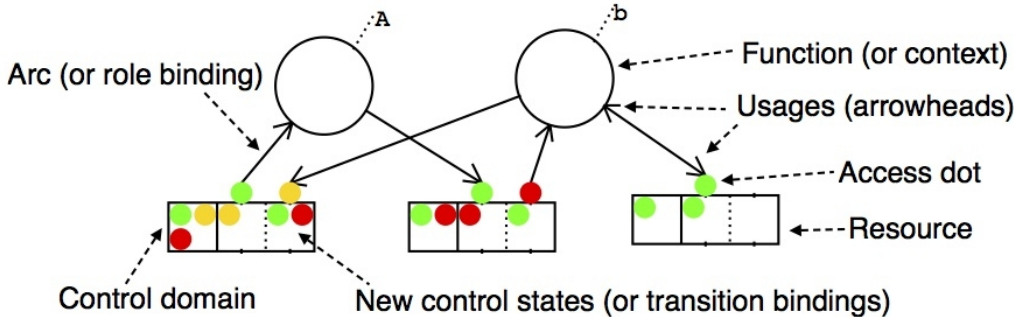
Many of these questions were being asked in the 1980s by Robert Babb (as they have been by others). His Large Grain Data Flow (LGDF) approach [7] considered the arcs as just repositories (essentially variables), and he used special annotations when they wouldn’t just be going from one node to the next, but might instead be accessed or modified again by the source node, for example. However, the notation became rather ungainly, nonportable, and vague, so his grad student, the present author, set about making a more well-defined, formal, portable, and scalable model. Out of this came the F-Nets computational model (originally called LGDF2) in about 1990, and in about 1992 was made the basis for a full-fledged formal graphical language that had various names (such as Software Cabling), but renamed to ScalPL (Scalable Planning Language) in about 2002. The rest of this document will explain the basis for these. Peer reviewed articles by the author for this work are available [8,9], and there is a complete book covering all these subjects in depth [10].

F-Nets Intro

An *F-Net* (short for Function Network) has nodes very similar to a dataflow network. Those nodes represent functions, and when certain conditions exist, they fire atomically: That is, they do all of their work if they do any, unimpeded and unaffected by any other concurrent actions, and that work is to produce results on some of its arcs as a function of the content (data) that it finds on some of its arcs. But the arcs do not go from one function node to another as they do in other dataflow diagrams. Instead, each arc goes between the function node (represented as a circle) and a *resource* (represented as a rectangle)—so an F-Net is a bipartite graph. Each resource can be considered as a repository for a token containing some data/information, called the resource’s *content state* (or data state). And although the arrowheads on the arcs can be considered to represent the flow of tokens much as they do in a dataflow diagram, it is now between the function and a resource, and it may have arrows on either, both, or neither end, depending on that flow, so the function may produce results to some of the same resources from which it gets its inputs.

That is, the content state of a resource can be *observed* by a function node connected to that resource with an arc having an arrowhead toward the function, and can be *replaced* by a function connected with an arc having an arrowhead toward the resource, or both observed

and replaced if the arc has arrowheads on both ends. (An arc may also have no arrowheads, for reasons to be clear soon soon.) These arrowheads are sometimes called *usages*, and both observe and replace when used together are sometimes called *update*, because there's no way to tell externally if that the function didn't just update the token in place. There's no situation in which a resource will ever possess more or less than one content state—the resource's initial content state is either explicitly specified or set to some well-defined default, and each firing can only observe or replace it—so the content state is generally considered to just reside on the resource, much like the value of a variable, and not considered a token.



Since (one) content state is always present on each resource, its presence or absence obviously cannot control whether an attached function will fire. Instead, each resource holds another kind of state, too, called it's *control state*, which actually is best visualized as a colored token (like a poker chip). Again, each resource can hold just one, initially a green one. Additionally, an F-Net diagram is drawn with one or more colored dots, called *access dots*, right where each arc connects to its resource. A function can fire only when all of its arcs are *ready*, meaning that the resource on the arc has a control state matching one of the associated access dots. When the function fires, the control state tokens atomically disappear from all of its resources, and then the function evaluates by (potentially) observing the content state of the resources for which it has observe usage, and based only on those observations, replaces the content state for all of its resources with replace usage (some of which may also have observe), and providing new control state (tokens) to all of its resources. So, a function node can be considered as a (partial recursive) function from the content states of its observable resources to the content states of its replaceable resources and the control states for all of its resources. (This is why arcs with no arrowheads (usages) can be useful: They will still have access dots, and their control states will still control and be replaced by each firing.)

That tells when functions can fire. There is also a *liveness constraint*, that if a function is (continuously) ready to fire according to these rules, then it eventually will fire. There is no fairness implied: If two different functions bound to a common resource are both ready to fire repeatedly, there's no rule that says they will take turns, the same one may win out every time and the other may wait forever (allowed by liveness because it's not continuously ready). There is a small additional amount of notation: A legend of all of the possible control states for a resource (called the *control domain*) is often shown in the left end of its rectangle as a set of colored dots (often labeled to indicate their meanings), and within the rectangle adjacent to each arc connection, a set of *new control state* dots is shown to indicate all of the possible control states which that particular node/arc may assign to that resource.

It is sometimes useful to consider an entity, called a *Director*, that makes the decisions about which function will fire, and which takes care of some of the busywork like destroying the control states when a node fires, as well as making optimal use of the computational platform. We would of course like it to exploit any efficiencies it can when doing its work, such as to get nodes evaluating concurrently if possible.

One more subtle but powerful F-Nets rule: Predictability

One such efficiency would seem to exist when a node has an arc without replace usage, and

all of the new control state dots for that arc/resource are the same color. We'll call such an arc *predictable* because, in such a case, as soon as the node fires, the Director will seemingly know what both the new content state and the new control state of the resource will be when that node has finished: The content state will be unchanged, and the control state will match the color of those dots. In such a case, then, there would be no real reason for the Director to wait for the node to finish accessing the resource before allowing other nodes to access the resource. For example, the Director could "cheat" and predictively assign the appropriate control state token to the resource, thereby potentially allowing other nodes to become ready to fire earlier, facilitating greater concurrency. The Director would also then have to ensure that the node didn't assign the control state again, since that could be after it had been further manipulated, but since the Director is involved in control state assignments anyway (to determine what should happen next as a result), it could just dispense with that operation. And, the Director would have to ensure that no other node would update or replace the content state if/while the original node was still observing it. That can be accomplished in many ways, such as by only allowing other nodes to fire if they will access the resource without replace usage, or by slyly giving a new concurrent node (with replace usage) a new version of the content state and then disposing of the original version (being observed by the original node) once that node is done with it.

It turns out these are very useful and portable optimizations. For example, if there are several nodes that just need to observe content state of a resource, this trick allows them to all do so concurrently while adding no extra rules or special constructs to handle this case. Similarly, if two nodes were passing a resource back and forth, with one replacing or updating its content and the other just observing these results, this trick would allow the first "creator" node to begin on its second iteration while the second "observer" node was still observing the results of its first iteration—i.e. double buffering, which could actually extend more generally to n-buffering in some cases—again with no extra rules or constructs.

Unfortunately, there's a catch: In general, because the functions represented by the nodes are partial, they might not produce any result in some cases. In fact, it is generally indeterminable from inspection whether the node will or won't eventually produce that control state token. So, we don't really know that it will leave the resource with that new control state, or none at all, in which case the optimization trick would have been in error. We remedy this very simply, by defining away that problem. That is, regardless of the function specified by the for the node, if the node has any predictable arcs, once the node fires, the resources associated with those arcs will (sooner or later) get assigned a control state—i.e. of the color specified by their new control state dots. So, the optimizations described above are indeed always possible (and, in some sense, eventually required) for any and all predictable arcs.

Real World Implications

It is fairly straightforward to efficiently implement F-Nets on real-world computing platforms with traditional programming languages, and to make each node function/program act completely independently of the others, functionally and as expected by its semantics, even with much concurrency present. The content states on the observe arcs are the program's inputs or arguments, and the content states of the replace arcs (sometimes overlapping), and the new control states for all of the arcs, are its results. To make this work, the program is only allowed to communicate with its environment through those resources/arguments, and is not allowed to maintain any internal state from one execution to the next (though holding it on a resource through an update arc is perfectly acceptable). And, the program is required to be deterministic (unlike some languages which, say, allow iterators on sets to process those elements in different orders on different executions). Some sort of extra operation (often in the form of a macro or function call) must be provided to allow the node to specify the assignment of a new control state to (the resource on) an arc, and this operation generally has the side effect of declaring that the program is done accessing the content state on that arc. So this

“new control state” operation can be considered a “return” statement for that one argument. The program will terminate automatically when the last argument is thus returned (since there is no other way for it to further affect its environment), and if it terminates explicitly before that, the rest are generally returned with some sort of well-defined default control state.

In low-latency environments (uniprocessors and shared memory/multicore), the content state of resources can be statically stored, while in high-latency ones, the Director can move those content states from place to place as needed, most sensibly between node executions. If a node has replace (but not observe/update) usage for a resource, there is no need to move the content state to it before firing, the node can just start with a new buffer for that content state, so especially when coupled with the predictability optimizations mentioned above, this behavior is effectively standard message passing. As for control state, it is rarely necessary to explicitly store control states for resources, or compare it with access dots: Instead, it is more common to associate a counter (sometimes called a “reasons count”) with each node which indicates the number of arcs on that node which are not ready at the current time, and these counts are efficiently manipulated by the Director: The assignment of a new control state to a resource is translated instead into decrementing the reasons count for each node having an arc that becomes ready as a result of that new control state. If such a count reaches zero as a result, the node fires, which consists of adding it to a run queue and incrementing the reasons counts of nodes bound to any of the same resources with arcs which were previously ready (to indicate that the control state has been removed from the resource).

This F-Nets model is simple enough to describe operationally (as we have done here) but also formal enough to describe in terms of a small set of mathematical axioms which describe how the execution of an F-Net, as a partial order or DAG (directed acyclic graph), relates to the F-Net itself. From these axioms (or just often common sense), certain forms of F-Nets can be proven to be deterministic (always producing the same execution DAG). Similarly, the potential points of non-determinism can also be identified—i.e. the cases where the next node to access (i.e. fire while bound to) any arbitrary resource is not completely constrained by the access dots and possible control states. These points can then be automatically instrumented, with extremely small overhead (e.g. recording a few bits per node firing) so that even a non-deterministic F-Net can be replayed deterministically. While this is useful during debugging, it also simplifies and optimizes recovery from failures. That is, by occasionally recording the state of certain resources and the determinism trace, the execution graph from that point can be rebuilt exactly as it was before without re-executing the entire system from scratch.

ScalPL

So F-Nets help with the basic issues of breaking a large system down into functions which are built with traditional sequential languages, and which individually act just as they would in a purely sequential context. Each one has a purely functional specification, describing the mapping from the content states on its observable arcs to the new content states of its replaceable arcs together with new control states for all of its arcs. They F-Net as a whole can be proven in pieces, by showing that each node does what it is supposed to (liveness) and only under the circumstances which it is supposed to (safety) based on the control states and access dots of the resources.

Still, this bare-bones model is far from providing all of the functionality and abstraction required for real-world architecture, design, and programming. There is virtually no support here for reusability, for data (linked) structures with dynamic memory allocation, for data parallelism or dynamically sized networks of any kind (including recursive), for stepwise refinement (top-down design), or for object-oriented techniques. Even so, all of these things (surprisingly perhaps) fit very naturally within and around the F-Nets model, as demonstrated

by ScalPL. ScalPL remains fully graphical in nature, although there are certainly textual syntaxes which can be used to represent the same information (e.g. EScort).

Reusability

The node programs described above for F-Nets suggest that they are very dependent on the resources to which they are connected, and especially the control states for those resources, since it is the job of the node to assign a new control state as part of its execution, and it can only know the "right" one by knowing the control states being used and how, by other nodes. ScalPL remedies this by first separating each node into the function to execute at the node, called a *tactic*, and the node itself (i.e. circle with arcs) where it is to execute, called a *context*, which is labeled with the name of the tactic to execute there. Instead of referring to resources, the tactic code refers to named *roles* (as in, the role which each resource plays within the tactic), and instead of assigning new control states to each role, the tactic performs a named *transition* to the role (as in, this is the transition to perform to the control state). The arcs from the context, now called *role bindings*, are then labeled with the role names, and the new control state dots, now called *transition bindings*, are labeled with the transition names. In this way, a single tactic can be used in many contexts, each perhaps bound to different resources with different control states, and the role bindings and transition bindings associate the roles and transitions (respectively) used within the tactic to the resources and new control state dots for that context. Each tactic is typed with its roles (names), and for each, its usages, the content domain (data type), and transition domain (names).

Higher-level modules

In ScalPL, the F-Net-like diagrams are called *strategies*, and strategies and tactics are referred to collectively as *plans*. As described so far, the contexts within a strategy each correspond to a tactic, but strategies are parameterized so that they, too, can fit into a context in another strategy, to form hierarchical strategies. That is, a strategy also has roles, and for each, usages, content domain, and transition domain. Within the strategy, each role corresponds to a special resource, called a *formal resource*, which only attains an initial control state when the associated role binding in the parent is ready, and is made to share content state with the resource in its parent bound to that role, and for which certain control states (colors) are identified as performing a corresponding transition to that resource in the parent whenever they are attained. Simply put, these (and a few other) features allow a strategy to act as much or as little like a tactic as is desired: It can be functional or nondeterministic, can maintain state or not, can fire atomically or may access some or all of its roles/resources multiple times while others are accessed once or not at all, etc. The bottom line is that, when creating a design, each context in the higher-level strategies may represent plans with quite complex behavior, then those can be constructed of simpler and simpler strategies, etc., until the lowest level strategies use only tactics within their contexts.

Dynamic Plans, Patterns, Templates

In F-Nets, contexts can do some work before initiating a plan. For example, by leaving a label (of a plan) off of a context, and instead including a *`$plan`* role binding on the context, the context will first observe the resource on `$plan` to get a plan, and then initiate that plan within the context. Such a plan can even initiate itself within one of its contexts, recursively, with special constructs to terminate the recursion at the proper time. Another important example of work that a context can do before initiating a plan is replication. That is, if a context has one or more roles with names prefixed by "`*`" or "+", called *dup* bindings, then before initiating the plan, the context will observe those roles to obtain an integer (or range of integers) from each, and then will replicate the context that many times (in a sensible way) and initiate the plan in each one, while providing each with a unique integer within that range.

Arrays, Dynamic Memory Allocation, Data Structures, Data Parallelism

While the resources described so far have been *scalar* (zero dimensional), with just one

element ever accessed, in general, resources can have any number of dimensions, with effectively infinite elements in each dimension. Each *element* possesses its own content and control states, though the content domain, control domain, and initial content state of the resource applies to all of its elements. There is no implied spatial relationship between elements (e.g. based on the relationship of their indices in each dimension). To facilitate concurrent operations on different elements, a context can be provided indices (individually or in sets or ranges) in so-called *binding modifiers* on one or more role bindings, and these (like the `$plan` and dup bindings in the previous paragraph) will be observed by the context before initiating the plan, and other roles will be bound to only those specified resource elements. Because this occurs after any replication takes place (from dup bindings, above), each replicant context can be bound (for example) to a different element to result in data parallelism. Also, since arrays have infinite elements, simply binding one which has not been previously used is essentially the same as dynamic memory allocation, so indices into an infinite array can effectively be used as pointers. Other binding modifiers exist to shift all array indices in one or more dimensions by a particular amount, and/or to permute the indices to effectively transpose the array in place.

Object-Oriented Design

By default, when a plan is activated in a context, the resources within it are *instantiated*, and are eventually garbage collected when they are no longer needed, perhaps after the plan terminates (at which time it can be initiated again with new resources). By annotating some resources in any plan (which we'll call a *class plan*) with *instantiation levels* (shown as shadows behind the rectangle), and then processing that class plan through a special operation specifying that instantiation level, a new plan (essentially an instance of that class) is created with those resources instantiated, so their content and control state will be maintained through different activations of the instance plan, potentially in different contexts. As our terminology suggests, this is similar to acting like an object (class instance) in an object-oriented language, since the plan has a well-defined interface, maintains hidden local state which can only be accessed through the interface by internal operations (which can be considered as methods). A few more bells and whistles help to solidify this further. For example, methods can be declared, and *binding constraints* can be specified so that only roles use by each method need to be bound by a context when that method is "invoked".

Status and Plans

The work shown here demonstrates the promise and viability of this approach, but there is far more work to be done. Although these ScalPL constructs are a proof of concept that a practical software system can be constructed on this basis, software is a very manual and human-oriented process that can only be refined through trial and error, and that means real-world usage, which will serve to simplify tools, provide shortcuts, and add further refinements and extensions to the language. Arguing here that certain functionality is possible given certain constructs is insufficient to show that it will provide an acceptable and accepted approach.

For example, although GUI tools exist which understand ScalPL graphical syntax and use that to simplify and speed the construction of ScalPL strategies (i.e. diagrams), they can and must go far further to be sufficient for real-world programming. As software managers have already observed, any time a programmer spends thinking about where things should go on the page, and especially for routing lines (role bindings) to make them prettier or less confusing, is time that would not have been spent if using other programming techniques, even if there is additional value here as a result (in terms of documentation and understanding). Much of that can be avoided with clever automatic routing built into tools. Likewise, certain information is repeated, such as the usages of a resource's content state, both by the operations on its variables in the tactic programs and the usages (arrowheads) shown on the associated role

bindings in the strategies. Even if that level of verification of intention is useful in some cases (just as variable typing may be for sequential languages), and if explicitly showing the arrowheads can change program meaning intentionally (such as to manipulate predictability), there should be ways to automatically transfer the information from one form and location (e.g. the tactic program) to the other (arrowheads) for simple cases. And, although there is already a textual form proposed for ScalPL strategies, enough short-cuts and syntactic sugar should be provided to make it a truly practical alternative in simple cases where the full flexibility and expressibility of the graphical approach is not needed.

For simplicity, there are also inherent decisions made in this work which may prove to be obstacles, and which may be irreversible in any practical sense. For example, providing roles (arguments) that will act as constant streams seems largely contrary to much else here, like the transactional nature of tactics. There are ways of dealing with it, however; One can explicitly build queue objects fairly easily from infinite arrays of small elements, for example, and operating on one or more elements per transaction, to obscure from downstream tactics that the stream is actually being created piecemeal by independent actions. But this may actually be a more accurate portrayal of streams anyway, and it is currently assumed that optimizing this sort of implementation is superior to explicitly adding more features and commensurate complexity to the model itself to provide such functionality.

The F-Nets model is potentially simple enough to be used as the basis for standards that could be used web-wide to network together services, augmenting the simpler www/http interactions currently used. Some limited ScalPL functionality could also be of use there.

This work on defining and implementing ScalPL has existed primarily without funding for at least 25 years. Today's software requirements call for its further development more than ever. It is at least intriguing to consider how useful and powerful it could become with adequate development resources.

References

1. Kahn, G. (1974). Rosenfeld, Jack L., ed. *The semantics of a simple language for parallel programming* (PDF). Proc. IFIP Congress on Information Processing. North-Holland. ISBN 0-7204-2803-3
2. Gul Agha (1986). "Actors: A Model of Concurrent Computation in Distributed Systems". Doctoral Dissertation. MIT Press
3. Hoare, C. A. R. (2004) [1985]. *Communicating Sequential Processes*. Prentice Hall International. ISBN 0-13-153271-5
4. https://ptolemy.eecs.berkeley.edu/
5. https://web.wpi.edu/Pubs/E-project/Available/E-project-031210-134520/unrestricted/FinalReport.pdf
6. https://en.wikipedia.org/wiki/SISAL
7. R. G. Babb II and D. C. DiNucci, "Design and implementation of parallel programs with Large Grain Data Flow", in The Characteristics of Parallel Algorithms (ed. by L. H. Jamieson, D. B. Gannon, and R. J. Douglass). Cambridge, MA: The MIT Press, 1987, pp. 335-349.
8. David C. DiNucci, PhD Dissertation, 1991, Oregon Graduate Institute, of Science and Technology, Hillsboro, OR, “A Formal Model for Architecture-Independent Parallel Software Engineering”
9. David C. DiNucci, “Tolerant (Parallel) Programming with F-Nets and Software Cabling” in Proceedings of the Workshop on Software Engineering for Parallel and Distributed Systems (PDSE’97), Boston, MA, May 1997, pp. 198-209
10. David C. DiNucci“Scalable Planning: Concurrent Action from Elemental Thinking”, May 2012, ISBN 1475211163, https://www.createspace.com/3830688