Software Backplanes, Realtime Data Fusion and the Process Trellis

Michael Factor and David Gelernter

YALEU/DCS/TR-852
March 1991

Software Backplanes, Realtime Data Fusion and the Process Trellis

Michael Factor and David Gelernter

YALEU/DCS/TR-852
March 1991

Software Backplanes, Realtime Data Fusion and the Process Trellis

Michael Factor and David Gelernter

Yale University
Department of Computer Science
New Haven, Connecticut

1 Introduction

A software backplane is a stuctural framework that can be re-used for many different programs. Programmers start with the backplane, and plug modules into it; the backplane manages all inter-module communication, and provides an interface to the user and to a programming-tools environment. Plug-in modules must adhere to a standard interface format, but otherwise they may do whatever they like. In logical terms, each module is an active entity: all modules on the backplane are active concurrently—more or less as they are in the hardware domain.

Several well-known software technologies fall generally into this category. Rule-based and blackboard systems are the most important. This paper describes a new software backplane, called the Process Trellis.

Existing backplanes have been crucial to some domains, notably expert systems. But they have been largely peripheral to the main body of software research and practice. This will change, we believe. This paper argues that software backplanes will become an important mainstream technology in the future, because they stand at the intersection point of four crucial requirements. We need modular and reliable software; we need high-level development tools; we need to accomodate heterogeneity; and we need to focus the power of parallelism on hard problems—efficiently, but without requiring complex and painstaking performance debugging on the programmer's part.

Prognostications about software are of no interest in a vaccuum. We've investigated our hypotheses about software backplanes and the Trellis in the context of a particular domain, the realtime data fusion area. Broadly speaking, realtime data fusion requires the realtime synthesis of a "big picture"—an

analytic overview—from a (potentially) large and diverse collections of input streams. These streams might describe the condition of a hospital patient, a financial market, a scientific experiment, a factory, an airplane, a transportation network... Like the backplane method itself, realtime data fusion is regarded by many computer scientists as esoteric and peripheral, to the extent it is regarded at all. Like the backplane method, it is destined to assume (we believe) a more important role in the near future.

We'll address the general issue of software backplanes first (Section 2), and then the character of the realtime data fusion problem (Section 3). We then discuss our particular backplane, the Process Trellis.

The Trellis imposes a simple and uniform structure on complex, heterogeneous programs; and it does so in such a way that parallelism is inherent in the resulting software. But the Trellis's most important property is somewhat idiosyncratic: the Trellis represents a strategy for *embodying the "intellectual structure" of a problem domain directly in software.* The first step in building a Trellis is an analysis of the major factors that inform decision-making in the domain, and their logical interconnections. The structure of the Trellis is then copied directly, in a sense *transcribed*, from this analysis—as we'll describe. The *user interface* is copied in turn directly from this software structure.

The Trellis is implemented in C-Linda[1][CG89], and has been tested in a number of domains; our largest prototype, which we will describe briefly, is designed for patient monitoring in the intensive care unit. Other papers discuss particular aspects of the Trellis project in depth (for example the realtime scheduling algorithm [Fac90], the intensive care unit prototype [FSC91], the Trellis as a formal system [Fac91] and multiple-Trellis ensembles [FGS91]). Our intention here is to present an integrated overview, to discuss the project's implications and to relate this particular experiment to (what we believe to be) its highly significant context: the realtime data fusion problem, and the software backplane methodogy.

2 Software Backplanes

It's hard to design, debug and maintain complex programs. Research aimed at making these tasks a bit easier has tended to focus, in recent years, on formal specification techniques and object-oriented programming. ([EM90] is a good survey of what's new and trendy.) Software backplanes offer a different approach.

A software backplane is a framework that can be used in building many different programs. To build a particular application, the developer plugs an

[1] A registered trademark of Scientific Computing Associates, New Haven.

appropriate set of modules into the backplane. "Plug in" means different things in different contexts: it may involve the execution of some special integration and scheduling routines (as it does for the Trellis), or simple link-editing, or something else. Regardless, the "plugging in" process is simple, uniform and intuitive in concept.

All modules share the same interface specification (so they'll be compatible with the framework); but otherwise they may differ radically in structure and purpose. The backplane manages all inter-module communication and provides interfaces to the user and to a programming-tools environment.

Micro-reusability vs. Macro-reusability

The backplane idea (as represented by earlier architectures as well as by the Trellis) relates strongly to the widely-discussed goal of "software reusability". But in a sense, the backplane idea is "software reusability" turned inside out.

Software reusability as generally understood centers on the reuse of standard modules from application to application. Object-oriented program design is an important variant: it involves the re-use of class definitions, not merely as templates for whole objects, but as adjectives that impart previously-defined features to new objects. (The technique was introduced by Simula 67.)

Software backplanes promote "reusability" in the inverse sense. Instead of providing a bunch of components that you may assemble in any shape you wish, they provide the shape and invite you to populate it. In module-level reusability schemes, the same component may appear within radically different program structures. In the software backplane model, the same structure may be imposed upon radically different sets of components.

In this sense, techniques like object-oriented programming promote "micro-reusability," software backplanes "macro-reusability."

(Object-oriented programming offers guidance in how to build modules, but imposes no structure on the program as a whole. This point is often misunderstood, by people who believe that "object oriented programming" in itself offers some kind of answer to the all-important question "how should we structure software?" Object-oriented programming represents an important advance in building materials, but no contribution to program *architecture*. Software backplanes are exactly the reverse. It might be argued that object-oriented methods *do* provide guidance on program structure: a program should be structuctured as a collection of objects which communicate by invoking each other's methods. But as a software *architecture* this scheme is vacuous: it answers none of the hard questions. Which modules communicate with which others? When and how? How does the program as a whole communicate with its environment? *These* are the questions that software backplanes address.)

The backplane's (and macro-reusability's) advantages

Backplanes are good for reliability: they allow us to partition-off a significant part of the problem and implement it once, with the intent of re-using this part repeatedly. And the partitioned-off segment isn't some arbitrary chunk of code. It's a potentially difficult chunk, insofar as it deals with coordination (among the concurrently-active modules of the program, and between the program as a whole and the external environment). It's also the chunk that imparts shape to the whole, the organizing framework. Naturally, we focus significant effort on developing the backplane. Once we've achieved a reliable backplane, we've solved a significant element of any large-scale concurrent system.

By imposing a fixed organizing strategy on the program as a whole, backplanes allow subtantial investment in design, debugging and visualization tools optimized to that framework to be amortized over many applications.

Simple, standard, one-size-fits-all interface specifications promote heterogeneity: two separate modules can be developed in complete mutual ignorance, using radically different tools and techniques, and yet be assured that they'll be able to communicate. (Note how the heterogeneity of telephones increased when the U.S. phone system ceased to be a hard-wired whole, and turned into a backplane—the network—plus plug-in modules.)

And the backplane is inherently a *concurrent* structure. A successful backplane design can make explicit parallelism all-but-transparent: applications programmers build parallel programs with only the most general awareness that they are doing so, and without needing to master any new tools or techniques beyond the backplane itself. (Substantial Trellis applications, yielding efficient C-Linda programs that run on a large range of parallel machines, have been developed by applications programmers with no knowledge of parallel programming in general or Linda in particular.)

The macro-reusability of the backplane approach in no way conflicts, of course, with the micro-resuability promoted by object-oriented and other approaches. Just the opposite: a standard backplane strongly promotes reusable modules, by assuring that they can be incorporated as-is into many applications.

3 The Realtime Data Fusion Problem

We turn now to the application domain, which is a substantial topic in its own right.

Realtime data fusion refers, again, to the integration and analysis of a collection of incoming data streams. Software for realtime data fusion is a research field that is preparing for an imminent explosion. The reasons are simple. Ma-

chinery and organizations are getting more complicated: burgeoning floods of data are available to characterize their states, and the machinery itself is more sensitive to constant fine-tuning. Human users can't keep up now, and if present technological trends continue, they will fall further and further behind.

We can summarize the problem in terms of an ever-widening "control gap." The control gap is the gulf separating the *optimal* response to a time-critical situation from the *actual* response that human operators—struggling under their inherent data-processing limitations—can achieve. Realtime data fusion is an attempt to bridge this constantly widening gap.

For example: in intensive care units and operating rooms, clinicians must interpret and react to a complicated and diverse collection of unstoppable data streams. Unless the correct interpretation is available *fast*, it's useless. And technological trends are tending to make this hard problem harder. They tend to increase the volume, diversity and accuracy of the data that can be gathered, and the range of available responses to any given problem—while doing nothing to lessen the urgency of the required response, or to increase human data-processing capacity. In these situations, clinicians face the obvious difficulties of processing and interpreting masses of data correctly. They also face the more subtle problem of "fixation"—the natural human tendency to become biased towards an initial hypothesis and to ignore or misinterpret data to the contrary. Aircraft control involves similar problems: masses of data, time-critical decisions, a hard problem getting worse. Similar problems arise in the control of complex systems of all kinds—ships, factories, airports, power plants and so on.

Another example: consider the the masses of data produced by scientific sensing equipment and laboratory experiments. Data may need to be interpreted in realtime, either to allow for an immediate response (as in weather prediction)—or simply because, in a long-running experiment producing high volumes of data, once you fall behind you are stuck forever. Your data backlog grows without let-up, potentially overwhelming storage and cataloging facilities. It becomes imperative to extract *value* from the data, some idea of its *conceptual content*, in realtime. There are many scientific domains in which massive data-handling is a growing problem. For example, "data volumes generated by very large array or very long baseline radio telescopes currently overwhelm the available computational resources" [Brom89]. NASA's "earth observation system" is designed to generate a terabyte of data per day when it comes on line in the late 1990's [Sci90a]. A headline in *Science* summarizes the problem: "Learning to drink from a fire hose [Sci90b]." Realtime data fusion systems make it possible to present an accurate, high-level, pre-processed synopsis for further analysis instead of a mass of low-level data. Related problems occur in the monitoring of financial, economic or commercial information.

Realtime *expert monitors* in particular may represent the most important species of realtime data fusion software. An expert system uses heuristics in

problem domains where determinate algorithms are unknown or intractable; an expert monitor is a high-performance expert system, capable of functioning as a monitor in realtime. The data fusion systems we will describe are in fact "expert monitors." They don't rely on heuristics exclusively, by any means; but they draw freely on the techniques of applied AI.

4 The Trellis

We move now to the specifics of our project. We describe the Process Trellis, which is our particular software backplane for realtime data fusion applications.

Consider a series of separate decision processes. The processes are hierarchical; higher-level processes deal with broader or more general sub-problems. For example, each element in the lowest level of decision processes might be wired directly to external data sources. The next-higher levels might perform initial data filtering, trending and baseline calculations. Levels above this might recognize fairly narrow patterns. Still higher levels might recognize broader or more complex patterns, and top level processes might perform "meta" services— evaluating the reliability of certain aspects of the system, the applicability of the existing decision structure and so on.

All processes run concurrently. Each can be regarded as driving a meter that displays, at all times, the current status of the sub-problem it's focussed on — the latest input value in some data stream, the probability that a given diagnosis correctly describes the situation, and so on. These processes are logical black boxes with respect to the Trellis. As long as they conform to the Trellis interface protocol, each process can incorporate any kind of logic that seems appropriate. Processes may be statistical, analytic, heuristic or anything else.

Each module in the Trellis (a module is simply a decision process) continuously attempts to calculate a state based upon the states of inferior modules. When sufficient information is available from a module's inferiors — each decision process defines for itself what "sufficient information" means — it generates a new state. As a result, modules one level up may recalculate their own states, and so on. Values in this sense flow upward through the Trellis. Besides passively waiting for inferior modules to change state, modules may send queries downward through the Trellis, forcing their inferiors to generate updated state values. Queries in this sense filter downward.

We supply two types of "logic probe" with the system, a "write" and a "read" probe. We can touch any module with a write probe, thereby setting the state of the module we touch to any value we choose. In the default configuration described above, each module in the bottom rank has a permanently-attached write-value probe through which we pump new values into the system. We can

read any module's current state by touching it with a "read-value" probe. If the module has insufficient information to have a currently defined state, touching it with a read-probe causes queries to propagate down to each of the its inferiors. Eventually new data values arrive and a response is produced.

5 For Example

Our most substantial experiment with the Trellis to date is the prototype hemodynamic monitor for a post-operative intensive care unit (ICU). This monitor[2], which is intended ultimately for clinical trial, is still a basic research project. But it is a sizable and growing application and a useful test case.

Figure 1 is an excerpt, showing the relationships among the various levels of the hierarchy, some typical occupants of each level and their inter-relations. Figure 2 shows the entire program. (In fact, this figure is somewhat out of date; the current program is an ensemble of two Trellises, as we explain below. But figure 2 is nonetheless a useful picture of you might expect a substantial "real-world" Trellis to look like.)

Figure 1 shows how the structure of a process Trellis program mirrors the domain's logical structure. At the bottom level are interface processes that gather data from monitoring equipment or — in the prototype case — from the realtime patient simulator. *Raw BP*, for example, receives blood pressure values. At higher levels are processes that, like *BP* (blood pressure), calculate symbolic values and detect trends in the lower level's outputs. Higher levels, for example the level containing the *SVR* (systemic vascular resistance) process, compute values based on several lower-level data streams. The next level of processes, for example *Hypotension*, look for various "clinical scenes" — physiologic states whose presence, in a particular time ordering, enable the top level processes (for example *Tamponade*) to make diagnoses. (A detailed description of the knowledge representation issues of processes at this level is found in [CSG89].) Earlier versions of the prototype had a still-higher level of "meta" processes which could recommend courses of action (e.g., "perform an infectious disease workup") and summarize, either textually or graphically, the patient's condition.

Using the process Trellis shell, two researchers without any specific knowledge of Linda or of parallel programming (Drs. Cohen and Sittig of the Medical Informatics program) have written, debugged, and tested a majority of the processes in the current version of the prototype. The current ICU Trellis comprises roughly 27,000 lines of code; it has been tested (off-line) on real patient data, collected during open-heart surgery cases. (The program is intended for use not in operating rooms but in intensive care units; for this type of monitoring,

[2]joint research with Drs. Perry Miller, Aaron Cohen, Dean Sittig and Stanley Rosenbaum of the Anesthesiology Department and Medical Informatics Program at Yale.

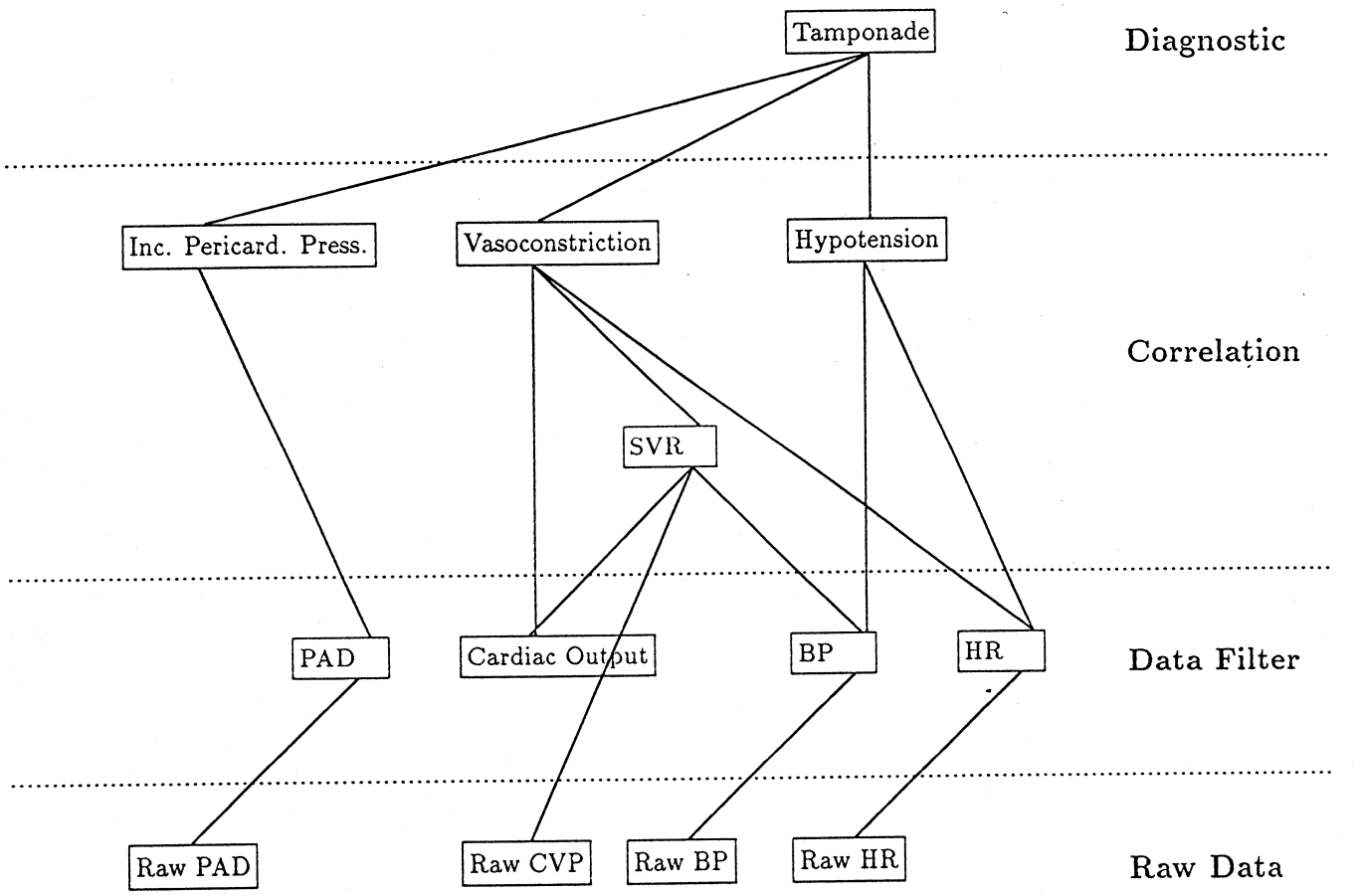

Figure 1: Prototype ICU Trellis: sub-set

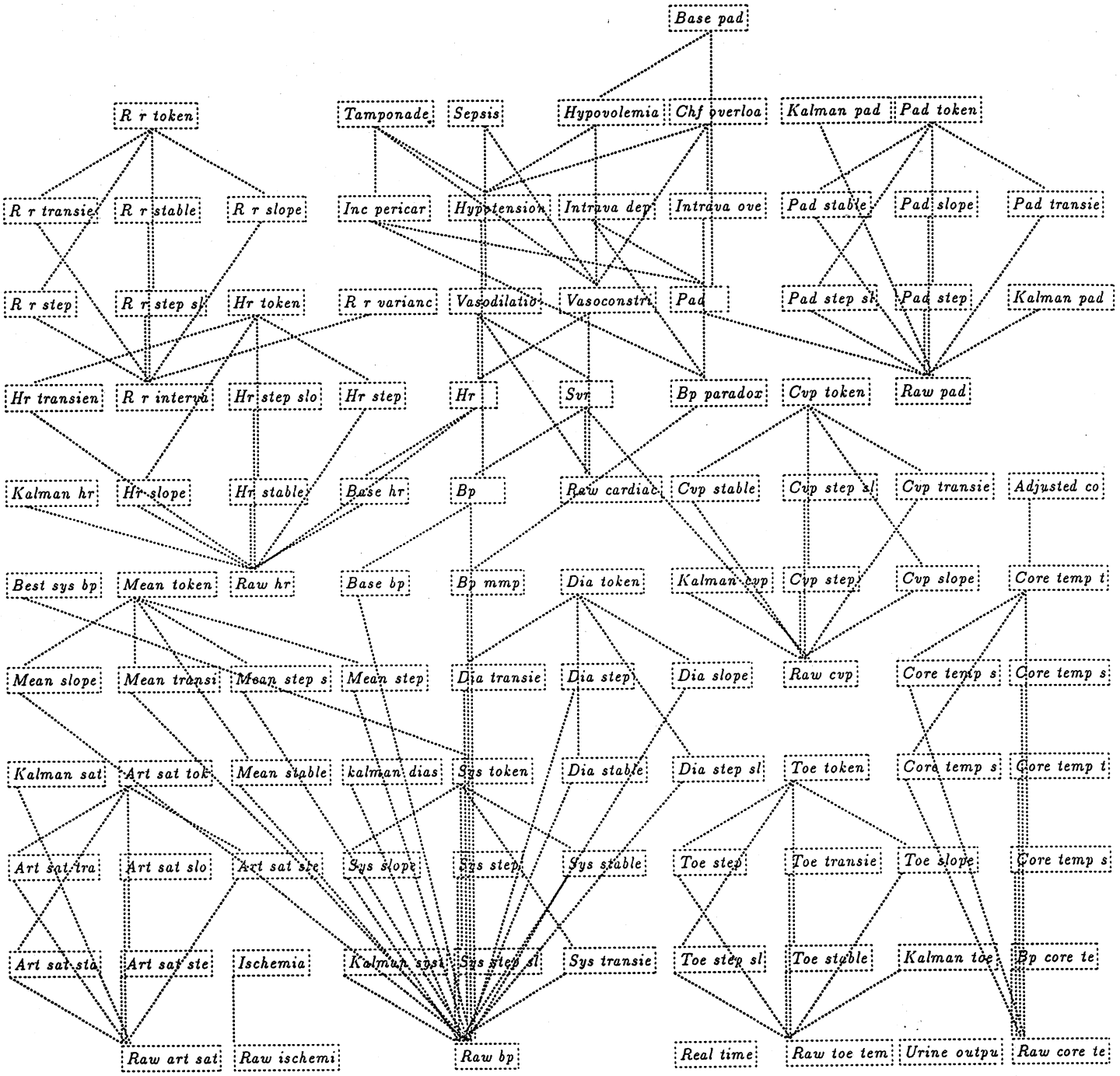

Figure 2: Prototype Trellis for ICU monitoring. (Details aren't important, but the overall scale and structure of the program should be clear.)

however, surgical and post-surgical ICU conditions are similar.)

Factor's thesis [Fac90b] describes another implemented Trellis (for the analysis of blood pressure waveforms), and two other Trellis designs (for financial markets and climate-data monitoring) produced collaboratively with domain experts.

The sections following fill out the Trellis picture by describing the interface, our parallel realtime scheduling technique, the role of probes, and our Trellis-building tool (the Trellis Shell). We turn first, though, to the central question.

6 Why?

What's good about the Trellis? Why is the Trellis a better structure than rules, blackboards, generic "object oriented programs" and other alternatives for realtime data fusion applications?

The designer of a complex application of this sort faces three key questions:

(a) *What is the intellectual structure of the problem domain?* When an expert evaluates the evidence and attempts to reach a conclusion about some particular problem, what "basic factors" does he need to consider? How do the basic factors relate to each other? If he were to capture his feel for the problem's structure in a mental checklist, what items would be on the checklist?

(b) *How should my program be structured?* What are the major activities that my program will need to carry out in order to solve the problem? How should my code be structured and modularized? Which modules communicate with which others, and how?

(c) *How should my program communicate with its users?* How will I get information in and out? What kinds of data will users want? This question is critically important for programs like realtime data fusion applications, which often know too much for their own good. An intelligible interface is crucial, to avoid burying the user in more data than he can use and thereby rendering the program useless.

Under conventional program-building schemes, these are three separate questions. In the Trellis approach they are *one* question. The Trellis reduces all three questions to the first one: what is the intellectual structure of the domain? Once you have solved this problem, you have solved the other two as well. The central facts about this architecture are: *The intellectual structure of the problem* becomes *the structure of the software*; and *the structure of the software* becomes *the structure of the interface.* In short,

Intellectual structure ≡ *Software Structure* ≡ *Interface Structure*

To express this identity in another way: a Trellis program focussed on some evolving real-world system isn't an external monitor, as the term is usually conceived—some piece of machinery, of arbitrary internal design, trained on the system of interest. Rather, it's a *model* of the real-world system, an *embodiment* or microcosm. A model of a special kind, of course: a model of the system *as abstracted by an expert observer*—not of the raw physical stuff. A model not of the patient, but of the intellectual problem of diagnosis; not of the powerplant, but of the intellectual problem of operating a powerplant efficiently; and so on.

We explain these statements and their import in terms of the following points. (1) In Trellis programs, there is a one-one mapping between *basic factors* or intellectual entities in the problem domain and *modules* in the program. (2) In Trellis programs, there is a one-one mapping between *logical connections among basic factors* and *communication paths* in the program. (3) In Trellis programs, modules are organized into a conceptual hierarchy that mirrors the conceptual hierarchy of the problem domain. There is an orderly and well-defined flow of information among levels. All modules are active concurrently. (4) Summing over these facts, we conclude that the problem structure is embodied in the program structure. We may accordingly use exactly the same structure for the *user interface* as well. The user interface may have exactly the same shape as the program. It may report one "piece of data" for each module in the program. It may arrange these piece of data in the same hierarchy as the program uses. In so doing, it captures the shape not only of the program, but of the intellectual structure of the problem domain.

Which modules?

Trellis applications are structured according to the following rule: identify the basic intellectual factors in the problem domain; provide one module for each. One-factor-one-module is a simple rule. It may also sound like an obvious one, but it's by no means typical of most approaches.

In conventional modularity schemes, the structure of the program follows the exigencies of some sequential problem-solving strategy. A conventional program aimed at data fusion would be coordinated by a driver routine. The driver might call a routine to get a new data value, then (perhaps) an "analyze signal" routine (which invokes a series of more specialized ones), then perhaps an inference engine of some sort, then a "display results" routine. The module structure imposed on the program text probably reflects the driver's problem-solving strategy. The routines to get data, to analyze signals, to draw conclusions and to display results might be grouped into four large, separate envelopes. Each envelope might then be hierarchically organized into subroutines, and so on.

In rule-based architectures, a single factor might be splintered over many rules [see e.g. DBS84]. Rule systems are "naturally modular," it's sometimes

argued, insofar as each rule captures a single fact about or aspect of the problem domain. But each rule is small in scope, "fine grained;" we can't use "three big rules," say, to capture the three major aspects of the problem, and then define a series of slightly-smaller rules to implement each "major rule," and so on. Each rule captures a single sliver of information, and rules make up a flat collection: one rule can't "contain" other rules. Rules are in this sense inappropriate for building a modular system.

Blackboard architectures comes closest to the Trellis's one-factor-one-module strategy (in principle; although in practice, the knowledge sources in a blackboard program are usually rules, and accordingly suffer from the same modularity problems that other rule-based systems face).

How are modules organized?

In Trellis applications, the logical connections between factors are expressed in the form of explicit communication links between modules. Again, the rule is simple: one logical connection, one inter-module communication link.

In conventional program architectures, on the other hand, communication within the program travels strictly up-and-down the organization tree; there are no explicit communication links. There are no explicit communiction links in object-oriented programs either. (Further, communication via method-invocation—as in object-oriented programs—is inherently a poor match to Trellis-style inter-module communication. Method invocation is the logical equivalent of procedure invocation: invoke a method; wait; receive a reply. Communication in the Trellis is *asynchronous*: when information is sent from one module to another, no reply is expected or required.)

In blackboard architectures, again, there are no explicit communication links. Communication takes place implicitly via the blackboard. Blackboard systems in this sense fail to *identify* intellectual structure with program structure in the Trellis's sense—the links being (in our view) part of the intellectual structure.

The Trellis's explicit communication links give it another significant engineering advantage over blackboards; the Trellis has a property we might call "local comprehensibility." "Local comprehensibility" means that, if we need to change or add modules, we need only understand the new module's "neighborhood"—the modules connected to it directly above and below. We don't need to understand the program as a whole. This makes it possible to envision enormous Trellises incorporating tens of thousands of modules, or more. A Trellis with ten thousand modules is a machine of staggering complexity. No one programmer could understand such a program in its entirety. "Local comprensibility" means that, notwithstanding, we can imagine such a machine being

methodically assembled, tested and put into service. In a blackboard system, on the other hand, interactions between knowledge sources are dynamic, and the effects of adding or modifying a knowledge source may be unpredictable.

The logical hierarchy

What we've described so far is different from conventional approaches and from other software backplanes. It's broadly similar, though, to many program structures that fall into the "specialist parallelism" category. (In [CG90], we divide parallel program structures into three basic categories; in "specialist parallel" programs, each process in the concurrent ensemble specializes in one aspect of the problem, and specialists communicate over the edges of a logical graph or network.)

But the Trellis isn't merely an arbitrary network. Trellis modules are arranged in a particular way, and they communicate according to a well-defined protocol.

Trellis modules form a *conceptual hierarchy*, with conceptually "higher level" modules dominating lower-level ones in the graph. Trellises have multiple input streams; we can identify each input stream with a bottom-level module in the Trellis graph. Each module in the Trellis depends, ultimately, on some subset of all input streams. One module is "higher level" than another only if it depends (ultimately) on a set of input streams whose size is the same as or greater than the other's.

Informally, then, a "higher level" module fuses more data streams than a lower-level one, or it applies a further stage of processing and refinement to the same set of data streams examined by a lower-level one.

Referring again to figure 1, "Vasoconstriction" depends ultimately on four data streams, "cardiac output," "raw CVP," "Raw Blood Pressure" and "Raw Heart Rate." SVR is lower-level: it depends on three streams. Tamponade is higher-level; it depends on five streams. "Increased Pericardial Pressure" is higher-level than "PAD" insofar as it applies a further stage of processing to the same set of input data streams (a singleton set in this case).

The formal ordering rule we've given is a constraint that determines when a module *cannot be higher-level* than another module. It leaves the developer free to make some determinations within this constraint: although module A may correspond to a larger set than module B, the developer may choose to rank the two at the same level in the Trellis graph, because they are conceptually similar. It's possible, though, to project the Trellis graph onto a regular structure that strictly captures our formal notion of hierarchy. We can draw a "process lattice" corresponding to any Trellis. In the process lattice (the powerset lattice over the set of input streams), each node represents one sub-set over all input streams,

and nodes are partially-ordered by set inclusion. Each module in the Trellis can be mapped onto the corresponding node (the node with the same input streams as that module) in the lattice. Many Trellis modules may be mapped to the same lattice node; many lattice nodes may have no corresponding Trellis module. The lattice represents a kind of synopsis or summary or overview of the corresponding Trellis. (Figure 3 is a process-lattice synopsis of the intensive care unit Trellis in figure 2.)

In section 3, we described the information flow pattern within the Trellis. This regular flow also strongly differentiates the Trellis from an arbitrary network-stuctured parallel program.

All Trellis modules are active concurrently. Hence they all exist and maintain a current state at all times. This fact distinguishes the Trellis both in engineering terms and "philosophically" from blackboard architectures.

In engineering terms, blackboard systems (unlike Trellises) are designed to invoke, repeatedly, the single knowledge source with the greatest marginal solution-finding value; much effort has gone into studying this inherently sequential scheduling paradigm [HR85].

The philosophical distinction is more fundamental. Most blackboard systems (and most rule systems—indeed, most applied AI systems in general) are designed to converge on the "correct answer" to an interesting problem. Trellis programs, on the other hand, are designed to capture some system's current state in its entirety. In the Trellis approach, designers are encouraged to supply a module for each and every interesting factor, no matter how specialized, rare or esoteric. This approach reflects our belief that the interesting problem of the near future is *not* how to deploy computing power thriftily, but how to squander it creatively. If a module focussed on some particularly rare and highly-specialized condition runs silently for ten years without ever seeing anything worthy of comment, and on day 3,654 alerts us to some rare condition that might otherwise have been overlooked—we're satisfied. (In fact, delighted.)

In Sum

It should be clear then that, given the identity of its modules, the connections among modules and the hierarchy in which modules are arranged, a Trellis represents an *embodiment of its problem domain.*

The implications for the important topic of interface design are clear. Because the program *embodies* the external system being monitored (is a *model* or *microcosm* of the external system), we can describe the state of the external system by describing the state of the program. Interface design isn't a separate project, independent of basic modularization and structural decisions; rather, having designed the Trellis graph itself, *we've already designed the interface—*

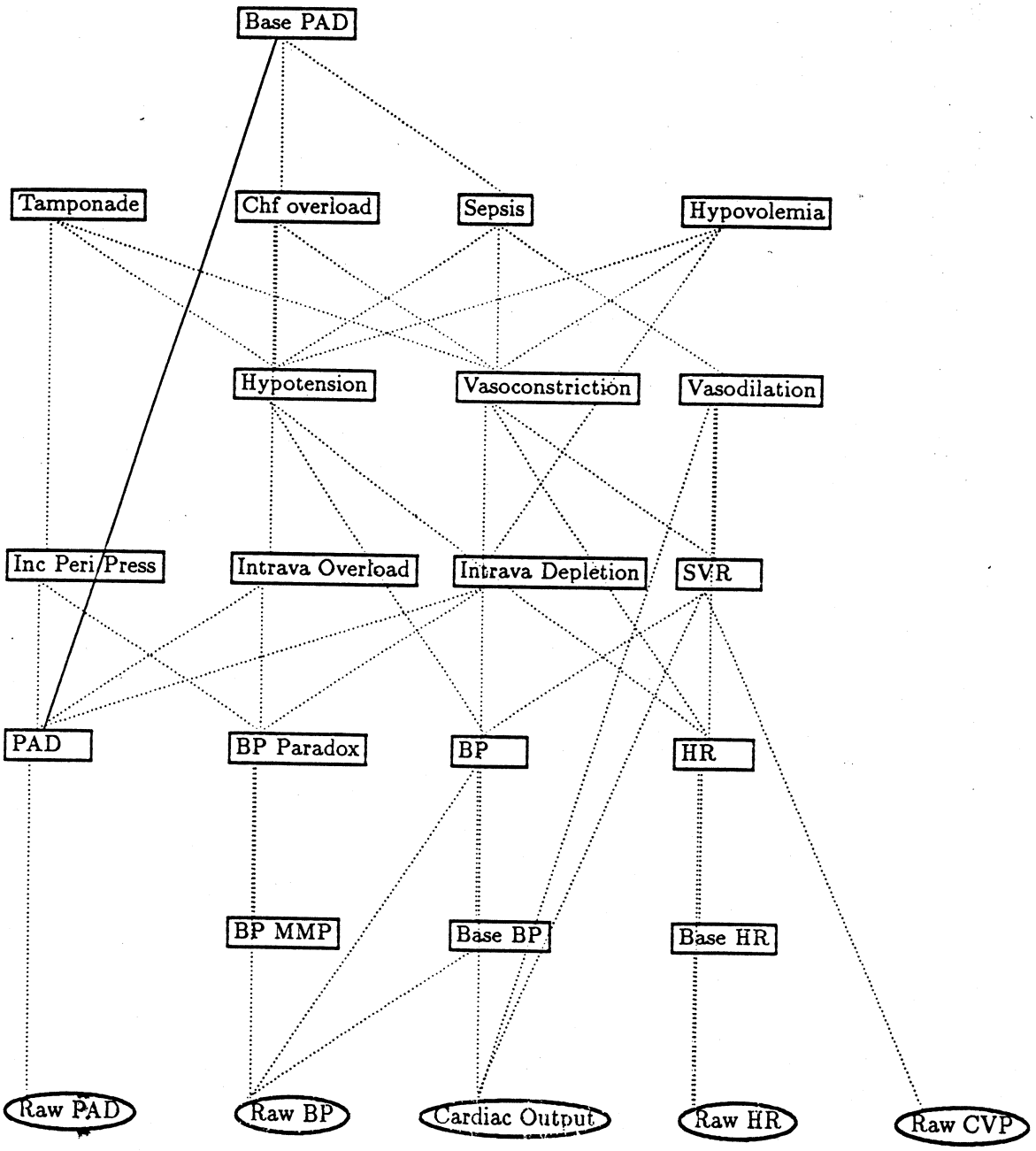

Figure 3: A subset of the ICU trellis...

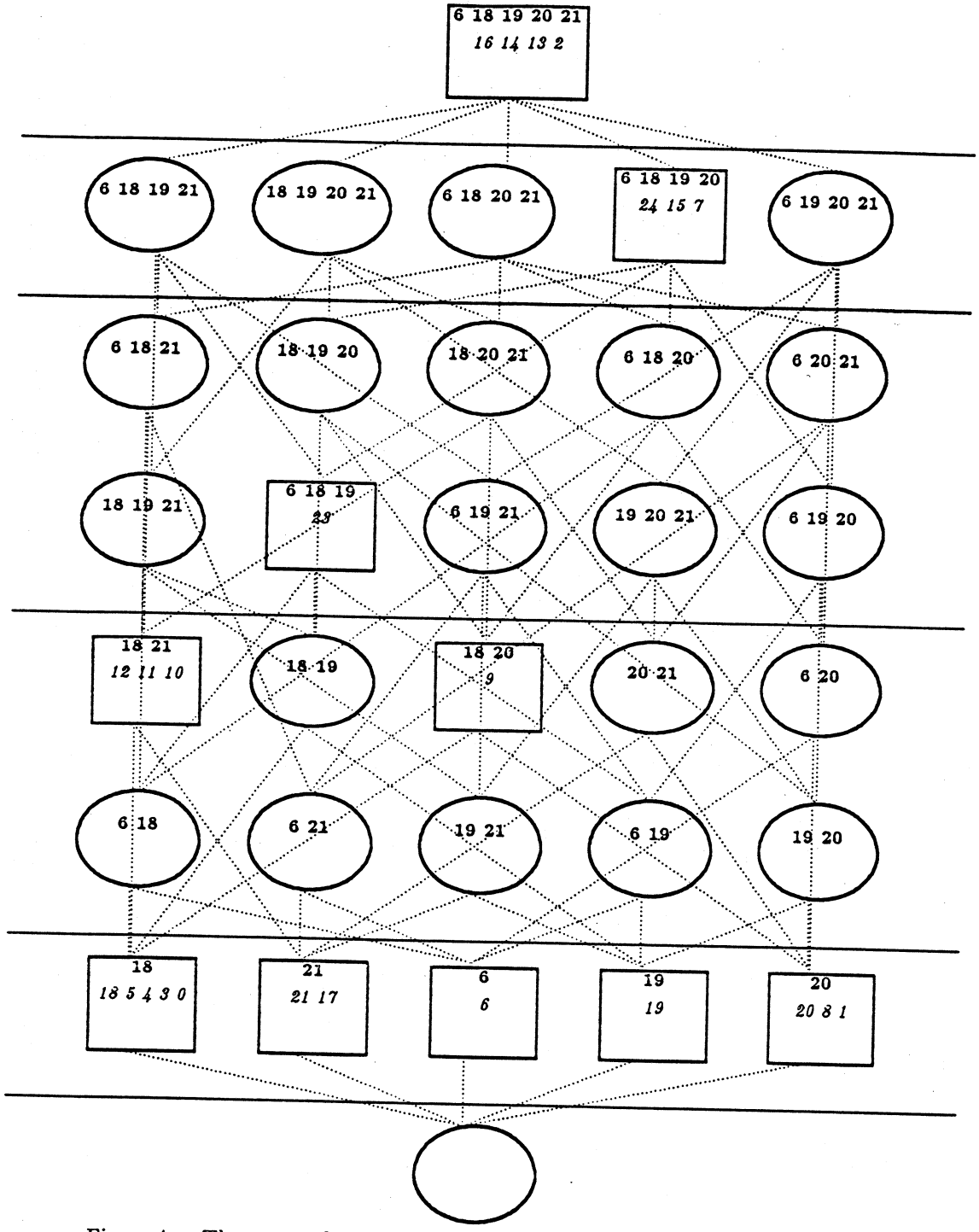

Figure 4: ...The process lattice corresponding to this trellis. Rectangles indicate lattice nodes that have associated trellis nodes. Ovals indicate lattice nodes with *no* corresponding nodes in the trells. The 10 rectangular nodes, suitably inter-connected, constitute the *lattice reduction* of the 24-node trellis shown in figure 3.

not the visual details, obviously, but the logical structure.

Our work on a graphics interface, discussed in the next section, focusses on visualizing the state of the Trellis program—and thereby, the state of the external system on which it's focussed.

7 Visualization

Our current display involves a receding plane modelled on the logical organization of the Trellis, and divided into "tiles." Each tile corresponds to a module in the Trellis; tiles closer to the viewer correspond to higher-level modules. When the value or condition monitored by a Trellis module is normal, the corresponding tile lies flat. As the monitored condition departs from normal, the associated tile angles forward and changes color. The user can select interesting tiles and set them up as separate windows, in which relevant current data about the corresponding module is displayed. (Figure 5).

The point in this style of display is, of course, to suppress uninteresting information, and to allow the user to focus his attention on the key points. The tile-plane display pursues this aim in several ways. All information about "normal" modules is suppressed (unless the user asks for it explicitly); flat-lying tiles integrate unobtrusively into the surrounding plane. Color change and tile-angle are twin indicators of a developing problem. Higher-level modules are closer to the viewer—whose attention will tend, accordingly, to be drawn first to serious "high-level" problems (because they are the closest and most prominent in the plane).

8 Realtime scheduling, and the frequency of the Trellis

Because all Trellis modules can execute concurrently, the Trellis architecture allows us to use parallelism to meet realtime constraints. Our goal specifically is to minimize the number of processors necessary to guarantee that the program analyzes incoming data within some time bound—that is, that the Trellis computes the full implications of incoming data, performs all the analysis of which it is capable, within an acceptable interval.

(In a previous section we claimed, of course, that the interesting computing challenge of the near future was the creative squandering of computing power. How does this claim accord with our goal of meeting realtime constraints with minimal processors? Creative squandering is, of course, very different from pointless waste. Computing power will certainly be cheap enough to allow us

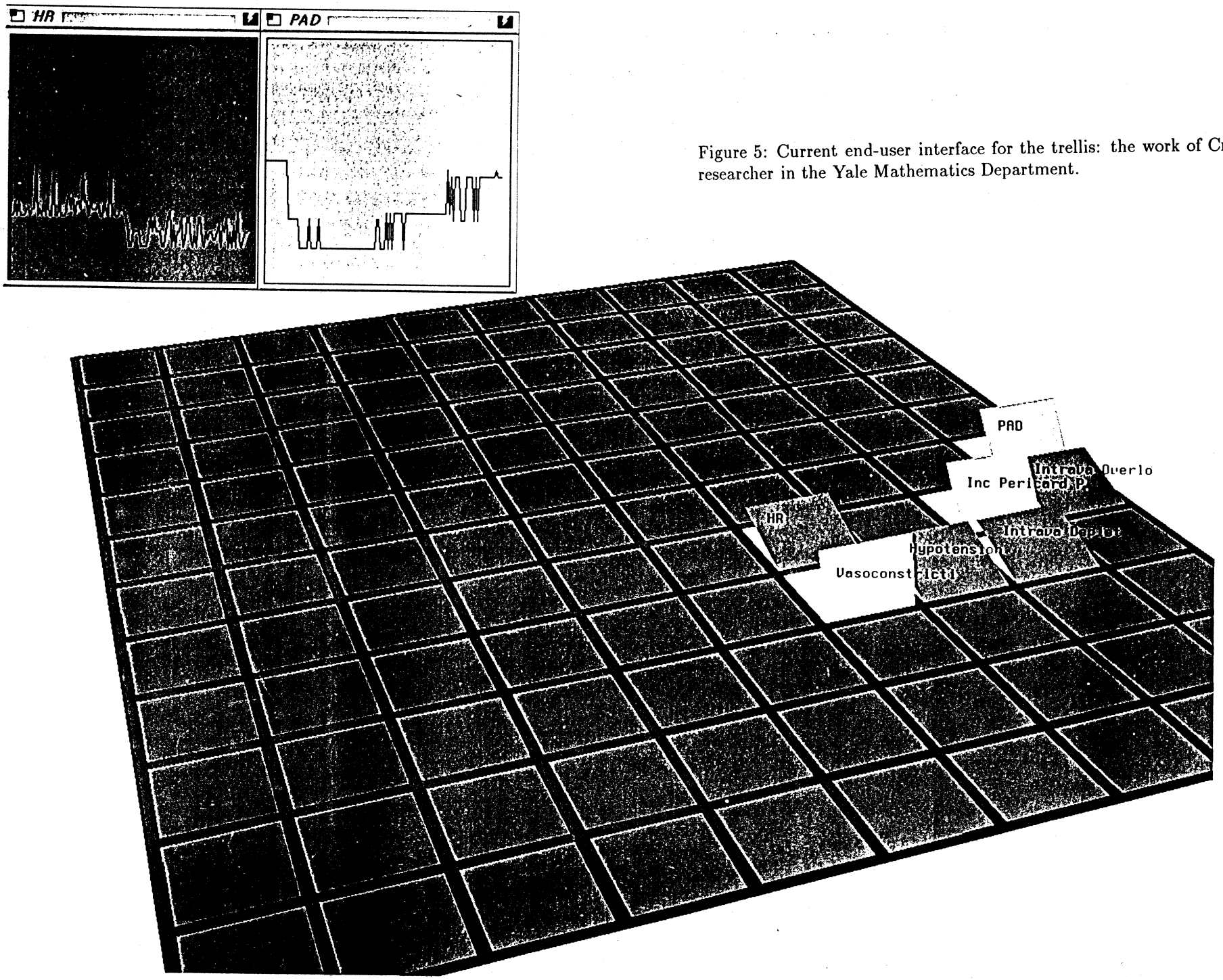

Figure 5: Current end-user interface for the trellis: the work of Craig Kolb, researcher in the Yale Mathematics Department.

to wager large sums against long odds, but few resources are ever cheap enough to waste indiscriminately. More concretely, we need to draw a clear distinction between our engineering goals for a particular software project and the boader intellectual context of which this implementation is one part. The Trellis is a working, useful piece of software as of *right now.* It's well placed to take advantage of the obvious ongoing trend towards cheap computing power. It's also engineered to be efficient on existing commercial hardware *today.*)

In the simplest implementation of a Trellis program, each Trellis module corresponds to (is implemented by) a separately-scheduled process. But we use a different implementation, more amenable to realtime scheduling. The Trellis is executed in a series of sweeps: all modules are updated for the ith time before any module is updated for the $i+1st$. Each sweep is executed by many processes concurrently. We create a fixed number of identical worker processes (typically one per available processor — each worker will run fulltime, essentially without blocking). We then use a scheduling heuristic to partition Trellis modules among workers. On the ith iteration, each worker updates the Trellis modules that have been assigned to it.

The Trellis shell implements this parallel-execution scheme. A collection of generic worker processes is created, one worker per processor. On each sweep, each worker updates its own set of Trellis processes.

We interpret our realtime constraint as follows: every input stream has an associated frequency—the maximum rate at which new values become available. We require that, whenever a new value becomes available on any stream, the entire Trellis has *already been updated* on the basis of the previous value. In other words, each module has recomputed its state *since* the previous value's arrival. (Other interpretations are possible, and some subtlety is involved in making these definitions precise: see [Fac89, Fac90].)

Our iterative execution scheme means that any running Trellis has an associated *frequency*: we can sweep through the entire Trellis, allowing every module to update its state, so many time a second. Note that, when we increase the number of processors on which a given Trellis executes, its frequency increases too—given more processors, each sweep goes more quickly, up to a limit determined by the longest-executing single module.

In order to satisfy our realtime constraint, we need only require that the frequency of the Trellis be greater than or equal to the frequency of its fastest input stream. This requirement defines the scheduler's task: the scheduler arranges Trellis modules onto (a heuristic approximation to) the smallest number of processors that allow the Trellis to run fast enough.

The scheduler presupposes an analytic model of program execution time, upon which it draws to predict the behavior of any given module partitioning. We have developed such a model; it has been tested and remains accurate

for synthetic Trellises of up to roughly 20,000 modules (the largest synthetic programs we can run under the current implemention on available parallel machines).

The Trellis scheduling problem is NP-complete; furthermore, no polynomial time approximation algorithm exists, since finding a feasible schedule is also NP-complete. Our scheduler is accordingly heuristic. We compared it to two other approaches, a first-fit algorithm and a simulated-annealing scheduler; our scheduler was clearly better than either (see [Fac90]). (It also ran, in some cases, roughly 5 orders of magnitude faster than the simulated annealing scheduler).

Where does Trellis execution belong on the loosely-defined spectrum from "hard" to "soft" realtime? The scheduler, and the analytic model upon which it is based, depend on certain timing information supplies as input: expected time to access local and global memory (a two-level memory model is assumed, as for example in Linda); communication delays as a function of number of processors; expected execution time of each module, and so on. The scheduler preserves the degree of "real-timeness" inherent in these input data.

To the extent that the numbers characterizing the machine's key performance parameters and the performance of the the modules individually are highly accurate, the scheduler's realtime performance guarantees are also highly accurate. To the extent that these numbers are approximations at some level of confidence, the realtime schedule is comparably approximate. Our scheduling approach is consistent, in other words, with whatever degree of hard-realtimeness is practical and desirable.

Related work

Our scheduler solves the following problem: given the value of a metric, in our case the make span (i.e. completion time), minimize the number of processors. Most other multi-processor scheduling work in contrast minimizes some metric on a fixed number of processors. For example, Stone [Sto77] and Lo [Lo88] minimize the total cost (i.e. maximize throughput) and Papadimitriou and Yannakakis [PY88] minimize the make span. Most work in real-time domains takes a similar approach, minimizing a metric, such as the number of processes missing their deadline, on a fixed number of processors [ZRS87,CC89,LPD88]. In many models this is the only approach available, since the worst case demands for computation and communication are unbounded. (They are of course *not* unbounded in the Trellis case.)

9 The power of probes

Probes provide a uniform mechanism for getting information into and out of a Trellis. They also allow us to string many Trellises together into a single ensemble: we can use probes as "data conduits" to feed information from one Trellis into another. We clip one end of the probe to a module in Trellis A and the other to a module in Trellis B. Depending on which way the probe is pointing, we can now pump data directly from A to B, or vice versa.

We usually think of multi-Trellis ensembles as hierarchies: a "bottom" Trellis receives input and refines it, then feeds the refined product to another Trellis, or to many others; they treat this refined product as raw input, and refine it further. In other words, we may string a probe from an upper module of A to a lower module of B: the A module's highly-refined information product becomes the B module's raw input. There are several ways to use probes in this fashion, and they differ in detail; but generally speaking, the probes function in every case like simple patch-cords conducting data from one Trellis to another.

Multiple Trellises are important for several reasons.

First, a Trellis operates (as we've noted) at some fixed frequency. Multiple Trellises allows us to build ensembles in which we need both "fast Trellises" and "slow Trellises." This flexibility is crucial. Suppose our data streams are "fast"—say, new values are available once per millisecond. Assume, also, that we have certain decision procedures that don't need to run frequently, but perform a lengthy computation (say, 100 milliseconds worth of computing) when they do run. We now have a problem: our fast streams will require attention once every millisecond; and we require that a Trellis completely update its state for each new set of input values. So, the Trellis we've attached directly to the fast streams must be capable of a complete update—every module executes once—once a millisecond. But if a module executing the long decision procedure is part of this same Trellis, we are unable to meet our requirements, because this one module will consume 100 milliseconds all by itself on those occasions when it needs to execute.

The solution is to build two Trellises. A "fast Trellis" is wired directly to the input streams. It's attached via probes to a "slow Trellis." The period of the slow Trellis is large enough to allow long decision procedures to execute to completion. The fast Trellis feeds partially-refined data into the slow Trellis. The ICU Trellis is, in fact, structured this way: the fast Trellis runs at about 100 Hz, the slow Trellis at 1 Hz. (Most modules are located in the slow Trellis. The structure is described in [FGS91].)

Multiple Trellises are useful for two other reasons as well, one obvious and one less so. It's obvious that multiple Trellises provide another, coarser level of modularity in Trellis applications, above the level of the individual Trellis ele-

ments themselves. In the weather and climate monitor described in [Fac90b] (a joint design study with researchers at Sandia Laboratory in Livermore), multiple Trellises are central for another reason: they will allow "custom Trellises" to be wired onto a main, public Trellis. The public Trellis performs basic data filtering and analysis; scientists with particular experiments to run may attach their Trellises to the main, public Trellis at any appropriate point. A custom Trellis performs information-processing for a particular scientist or research group. Such a Trellis might accept as input the highly-refined information product produced by high-level public-Trellis modules. Or, it might substitute its own filtering or analysis routines for some of the public routines. In that case, it clips its probes onto correspondingly lower-level modules of the public Trellis. At any time, arbitrarily-many custom Trellises may be wired into the public Trellis.

10 The Trellis Shell

The Trellis architecture is captured in a "Trellis shell". Programmers supply the shell with decision logic for each module; the shell builds the Trellis framework automatically, generating all the code necessary to support parallelism and handling realtime process scheduling. The shell provides in addition both a graphics and a menu interface to allow interactive invocation of probes, and Trellis program debugging.

To construct a Trellis program, the programmer uses a high-level tool that creates a database of process descriptions. The programmer specifies a process's name, the function it uses to calculate its state (this function will expect as arguments the state of this process and its inferiors), some information about the nature of the process's state, and a handful of miscellaneous routines. The shell is capable of compiling any subset of processes in the database (or the entire database) into a Trellis program.

A graphical program-development interface makes it possible for the developer to visualize a running Trellis program. (This interface is distinct from the end-user interface discussed above. The developer's interface is more detailed and allows a greater degree of control — and assumes, as a consequence, more knowledge — than the end-user's interface.)

The decision logic supplied by the user is written in C, and may contain arbitrary code. The Trellis framework generated by the shell uses C-Linda.

11 Present and Future

Future Trellis development focusses on refinement of existing Trellises and Trellis tools, and production of new Trellises. Trellis research focusses on three areas: (a) Sensor/actuator Trellises, (b) Turingware Trellises and (c) true bigness.

Sensor/actuator Trellises rely on a simple generalization of the Trellis information-flow protocol. In the Trellis as described, data values flow upward and "anti-data"—queries, in other words—flow down. But we could also interpret "anti-data" to mean *commands*. The lowest-level modules in a sensor/actuator Trellis may play either of two roles: some are connected to incoming data streams, as usual; others are connected to actuators. Such a Trellis is designed to monitor and (at least to some extent) to control a complex system or machine. Data values flow upward from the bottom; when they suggest that low-level, "tactical" adjustments to the mechanism are needed, the appropriate orders are issued by relatively low-lying Trellis elements, and communicated back downwards to actuator modules. (The actuator modules put them into effect by directly altering the state of the external machinery.) Data values that suggest farther-reaching, more "strategic" adjustments cause decisions to be reached and orders issued at correspondingly higher levels of the Trellis. Thus, tactical monitoring and control activity takes place near the bottom, relatively more "strategic" activity closer to the top of the Trellis.

This software model immediately suggests a biological analog. And the analogy (if it isn't pressed too hard) provides us in turn with useful new ways of thinking about the Trellis. In a sensor/actuator Trellis, "tactical" adjustments are planned and put into effect at low levels; we might add that

> While this immediate response is occurring, the same signals are being transmitted to higher centers for more elaborate analysis of their information content and for combination with signals from other types of receptors [Henn74].

The quotation describes the human sensory-motor system, which centers on a hierarchy that recalls the Trellis scheme: immediate "low-level," local responses at the bottom; more elaborate, global responses at higher levels.

The analogy in turn suggests that we might view the sensor/actuator Trellis as a kind of synthetic nervous system. (Note that we are fishing in rather different analogical waters from the ones frequented by the neural network research community. A neural network is an abstract model of the brain, not of the nervous system; not, in particular, of the data paths that connect the muscles to the brain. It's the *wiring of the nervous system*, the way information flows and decisions are made, not the *computational capabilities of the brain*, that we are interested in.)

It follows that, when a factory, powerplant, transportation system or whatever is equipped with a sensor/actuator Trellis, we might look at the resulting ensemble as a kind of "robot in the large." Robotics in the broadest sense studies "synthetic organisms," each one capable of responding as a *whole*, as a single integrated system, to its environment. In practical terms, Robotics as presently conceived focusses on issues of vision, navigation, physical manipulation and planning that are associated with mobile, more-or-less anthropomorphic robots. But there's no reason in principle why we can't regard an entire power plant (say) as a robot as well.

What we gain by doing so is a well-defined goal: to think of the building, factory, trasportation network or whatever not as a collection of separate systems (each, perhaps, *individually* computerized and quasi-intelligent), but rather as *a whole*, capable of bringing many systems to bear on the solution of a single problem and (importantly) of presenting a *single integrated interface* to the users. Thus, suppose computer scientists worked not in buildings but in macro-robots. We'd expect to be able to log on to "the building." We'd expect to able to pose problems to the building as a whole: we need to move three new people with phones and workstations to the fifth floor; propose some people-moving, furniture-moving and wiring plans. We'd expect to be able to ask questions: how much did you cost to operate last month? We'd expect to be able to integrate directory, people-finding and messaging services under the rubric of the building itself: Please find Fruitford and let him know that I'm in a meeting in room 300. Some of the required software relates directly to the Trellis, other pieces fit elsewhere. But the organizing framework and the "macro-robot" approach derive directly from the Trellis.

Turingware Trellises. We use the term "Turingware" to describe ensembles in which processes and people intermingle freely and "anonymously:" when some element (either a human or a software element) of the ensemble interacts with another, the first element doesn't know or care whether the other element is a person or a process. We discuss the idea in general and give some examples in [CG90].

The Trellis seems like a particularly appropriate framework for experiments with Turingware. It's easy to conceive of problem domains in which high-lying Trellis elements ought to be people, not software. Supposing we've identified some part of a Trellis problem that a person can solve more effectively than software, we can integrate that person "transparently" into the Trellis. He receives upward-flowing data (on his computer display, presumably); his responses are mailed upward in turn to the elements above. Queries or instructions may arrive from on high, to be processed and passed downwards. The Trellis allows us to impose a clear, simple and appropriate organizational strategy on a potentially far-flung and complex mess of people and programs.

True bigness. The Trellis is one scheme for organizing massively-parallel

asynchronous programs. Our tools and the architecture itself were designed with massive Trellises (on the order of tens of thousands and more) modules in mind. Tests of synthetic twenty-thousand module Trellises show them performing predictably and effectively on small current-generation parallel machines.

In very large Trellises, many modules will be similar or identical instantiations of a template: a transportation network will have a large number of segments all of which must be monitored separately and continuously, but the logic required will be basically the same in all cases; hence we may have, say, a thousand separate segements, each involving an identical 10-module sub-Trellis. The complete program, of course, will involve other differentiated modules as well. Scientific data gathering (in astronomy or weather domains, for example) may involve hundreds of data sources, to most of which essentially the same set of trends-detection filtering modules may be attached.

Of course, many Trellis domains will require a large collection of separate and distinct modules. Our medical collaborators estimate that the ICU Trellis will incorporate several hundred modules, most distinct in design, when it's finished; but this Trellis focusses only on hemodynamic monitoring. A general-purpose Trellis for monitoring surgical and intensive care patients in all areas would entail thousands of distinct modules. Beyond the inherent breadth of the domain, the complexities of developing reliable heuristic decision procedures will also tend to multiply modules. The Trellis structure can easily support multiple modules devoted to *the same diagnostic function.* We might test a new septic-shock diagnosis module by installing it alongside the old module; the old module continues to drive the display, but a write probe attached to the new module deposits values in a file, for performance comparisons against the old module. (It's also interesting to envision Trellis modules that are developed by experts in specialized areas, then disseminated to Trellis programs nation-wide.)

Conclusions

We believe that out Trellis work to date strongly supports the contention that software backplanes are a powerful tools. We've mentioned the fact that the bulk of our intensive care unit Trellis was constructed by programmers who had no knowledge of parallelism. There's a more general point as well: readers will be hard-pressed to adduce many parallel applications that involve more than one hundred separate and distinct activities, as the ICU Trellis does. They will be still harder pressed to find comparably complex parallel applications that meet realtime constraints. The mere fact that such a complex application was methodically designed, assembled, tested and analyzed by a small research group speaks strongly for the power of the backplane technique in general, and the Trellis in particular.

Acknowledgements

Our thanks to Craig Kolb of the Yale Mathematics Department for his work on the Trellis visualization, and to Drs. Perry Miller and Dean Sittig of the Yale Medical Informatics Program. This research is supported by the Air Force Office of Scientific Research under grant number AFOSR-91-0098.

References

[Brom89] *The Federal High Performance Computing Program,* Sept. 1989. Appendix A: Summary of grand challenges (49-50).

[CC89] Houssine Chetto and Maryline Chetto, "Scheduing Periodic and Sporadic Tasks in a Real-Time System," *Information Processing Letters,* 27,30(Feb 1989): 177-184.

[CG89] N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM,* April 1989(32,4):444-458.

[CG90] N. Carriero and D. Gelernter, *How to write parallel programs: A first course.* MIT Press (Cambridge, 1990).

[CSG89] Aaron I. Cohn and Dean F. Sittig and David H. Gelernter and Stanley Rosenbaum and Michael Factor and Perry L. Miller, "Sequential Clinical 'Scenes': A Paradigm for Computer-Based Intelligent Hemodynamic Monitoring, in *Proc. SCAMC 89* (Nov. 1989).

[DBS84] Randall Davis and Bruce G. Buchanan and Edward H. Shortliffe, "Production Rules As a Representation for Knowledge-Based Consultation Program, in William J. Clancey and Edward H. Shortliffe, *eds., Readings in Medical Artificial Intelligence: The First Decade,* Addison-Wesley (1984): 98-130.

[DS83] Randall Davis and Ried G. Smith, "Negotiation as a metaphor for distributed problem solving," *AIJ,* 1(20), (Jan 1983):63-108.

[EGR91] C.A. Ellis, S.J. Gibbs and G.L. Rein, "Groupware: Some issues and experiences." *Comm. ACM* 34,1(Jan. 1991):38-58.

[EHL80] Lee D. Erman and Fredrick Hayes-Roth and Victor R. Lesser and D. Raj Reddy, *The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty, ACM Computing Surveys,* 2(12) (June 1980): 213-253.

[EM90] Erin E. Murphy, "From an art to a science: Software R&D." *IEEE Spectrum,* 27,10(Nov. 1990):44-46.

[Erv90] R.D. Ervin, "Intelligent Vehicle-Highway Systems: Sooner and Later." UMTRI [Univ. Michican Transportation Research Institute] Research Review, 21,2 and 3: Dec. 1990, pp. 38-39.

[Fac90] Michael Factor, "The Process Trellis Software Architecture for Real-Time Monitors," *Proc. Second ACM/SIGPLAN Symp. Parallel Programming,* (Mar. 1990).

[Fac90b] Michael Factor, "The Process Trellis Software Architecture for Parallel, Real-Time Monitors." Yale Univ. Dept. Comp. Sci. PhD. Diss. (Dec. 1990).

[Fac91] Michael Factor, "The Process Trellis: A Formal Description." in *Proc. PARLE 91* (to appear).

[Fag80] Lawrence Fagan, *VM: Representing Time Dependent Relations in a Medical Setting,* PhD. Diss., Stanford University (June 1980).

[FG89] Michael Factor and David H. Gelernter, "The Process Trellis: A Software Architecture for Intelligent Monitors, *Proc. IEEE Int. Workshop on Tools for Artificial Intelligence.* (Oct. 1989).

[FGS91] M. Factor, D. Gelernter and D. F. Sittig, "Multiple Trellises and the Intelligent Cardiovascular Monitor." *Yale Univ. Dept. Comp. Sci. Tech. Report* (Jan. 1991).

[FL77] Richard D. Fennell and Victor R. Lesser, "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay-II", *IEEE Transactions on Computers,* 2(C-26), (Feb. 1977): 98-111.

[FSC89] Michael Factor and Dean F. Sittig and Aaron I. Cohn and David H. Gelernter and Perry L. Miller and Stanley Rosenbaum, "A Parallel Software Architecture For Building Intelligent Medical Monitors," in *Proc. SCAMC '89* (Nov. 1989).

[Gel89] D. Gelernter, "Information management in Linda," in M. Reeve and S.E. Zenith, *eds., Parallel processing and artificial intelligence.* J. Wiley (1989):23-34. (*Proc. AI and Communicating Process Architectures* (London, 1989).

[HarSS89] .H. Harrison, J.J. Shilling and P.F. Sweeney, "Good news, bad news: experience building a software development environment using the object-oriented paradigm," in *Proc. ACM Conf. on Object Oriented Prog.* (Oct. 1989):85-94.

[Henn74] lwood Hennneman, "Organization of the motor systems — a preview," in Vernon B. Mountcastle, *ed.,* Medical Physiology Vol. 1 C.V. Mosby Company (Saint Louis, Thirteenth Edition 1974).

[HR85] Barbara Hayes-Roth, "A Blackboard Architecture for Control," *AIJ,* 2(26) (July 1985):251-321.

[HR87] Barbara Hayes-Roth, "Dynamic Control Planning in Adaptive Intelligent Systems," *DARPA Knowledge-Based Planning Workshop, 1987*: 4.1

[Lo88] Virgina Mary Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Computers,* 11(C-37), (Nov. 1988): 1384-1397.

[LPD88] Victor R. Lesser and Jasmina Pavlin and Edmund Durfee, "Approximate Processing in Real-Time Problem Solving," *AIM,* 1(9) (Spring 1988): 49-61.

[PY88] Christos H. Papadimitriou and Mihalis Yannakakis, "Towards an Architecture Independent Analysis of Parallel Algorithms," *Proc. 20th Symp. on the Theory of Computing,* (May 1988): 510-513.

[Sci90] *Science,* 248,11 (May 1990).

[Sto77] Harold S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. SE* 1(SE-1) (Jan 1977): 85-93.

[ZRS87] Wei Zhao and Krithivasan Ramamritham and John Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems, *IEEE Trans. SE,* 5(SE-13), (May 1987): 564-577.