computation

coordination

Integration

or

Separation

Coordination Languages and their Significance

David Gelernter
Nicholas Carriero

We can build a complete programming model out of two separate pieces—the *computation model* and the *coordination model*. The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The coordination model is the glue that binds separate activities into an ensemble. An ordinary computation language (e.g., Fortran) embodies some computation model. A coordination language embodies a coordination model; it provides operations to *create* computational activities and to support *communication* among them.

Our approach to coordination has been developed in the framework of a system called Linda.™ Linda is not a programming language. Kahn and Miller write that "Linda is best not thought of as a language—but rather as an extension that can be added to nearly any language to enable process creation, communication, and synchronization[27]." We would rather say that Linda is a coordination language. It is one of two components that together make up a *complete* programming language. (The suggestion that traditional programming languages are *incomplete* is intentional.)

A computation model and a coordination model might be integrated into a single language. They might also be separated into two distinct languages, in which case programmers choose one of each: one computation language plus

ILLUSTRATION: DENNIS AHLGRIM

one coordination language equals a complete programming system. Kahn and Miller contend the first alternative is better. The heart of their comments is the observation that:

> Both camps are striving for uniformity, but of different sorts. The CLP [Concurrent Logic Programming] camp strives for uniformity within a language while the Linda camp strives for uniformity across languages [27].

We believe that the second alternative is better. We also believe that the distinction between the "integration" and the "separation" approach goes well beyond the (admittedly important) points raised by Kahn and Miller. It involves a whole range of pragmatic issues, and some deep questions about the nature and likely evolution of programming environments besides. We advance a series of claims in response to Kahn and Miller's argument for integration:

1. Asynchronous ensembles are *the* dominating intellectual issue in the emerging era of computer systems research—the era of dime-store processors and densely interconnected computer jungles. Diversity among an ensemble's elements—diversity with respect to language, hardware platform, physical location, even basic computing model—will be nornal in the new era.
2. The fundamental problems posed by ensembles—the problems of coordination among active agents—are best understood as *orthogonal to* the problems of computation, as addressed by conventional programming languages.
3. We can (and ought to) define "general purpose coordination languages" on analogy with general purpose computing languages. These coordination languages support a full range of ensembles, from parallel applications through distributed systems through time-coordinated ensembles and a range of others. The problems of ensemble building in general, in other words, constitute a well-formed and important intellectual unity.

If these claims are born out, they make a compelling case for separation.

Next, we briefly discuss the first (and least controversial) point. In the following sections, we explain our basic claims with respect to orthogonality and generality, discuss Kahn and Miller's case for integration, then present the argument for separation.

Ensembles and their Significance

We will define an *asynchronous ensemble* (ensemble, for short) as a collection of asynchronous activities that communicate. An activity is a program, process, thread or any agent capable in principle of simulating a Turing Machine. It could be a person; it could be (recursively) another whole ensemble.

Computations that are structured explicitly as sets of communicating processes will (obviously) be ensembles. So we have included parallel applications (designed to run fast on many processors), distributed systems (designed to manage physically dispersed hardware) and many operating systems (structured as ensembles in order to cope with the asynchronous simultaneity of devices). But the ensemble category is far broader. The definition does not restrict communication to communication-through-space—a collection of programs running during disjoint intervals and communicating *through time*, generally via a file system, constitutes a highly significant kind of ensemble. When a computation and a person communicate, we have another important type of ensemble. In short, ensembles are fundamental and ubiquitous in computing.

An ensemble is a natural breeding ground for heterogeneity. A program that needs contributions from different machines, or different computing models (i.e., synchronous and asynchronous parallelism), will naturally be an ensemble. Given their strong modularity, ensembles are a natural medium for multilanguage applications as well.

Finally, ensembles are the best-adapted inhabitants of the evolving hardware environment: the densely intertwined computer jungle that is taking root everywhere. In this developing environment, computations will rarely stay cooped up inside a single computer. They will interact with—draw services from and supply services to—other computations on other machines. Users will regularly choose to focus whole skeins of autonomous computers on a single problem.

Ensembles of all sorts dominate the near-term future of computing. Figuring out how to build and understand them will become (if it is not already) the central problem of systems research.

Basic Claims

Building ensembles equals coordinating separate computations. There is nothing new about our contention that ensembles and coordination are important. Kahn and Miller's own work on open systems and their linguistic requirements [22], along with other work in the open systems area (e.g., on Actors [2]), has played a leading role in bringing these issues to the attention of systems researchers. Our (more controversial) claims are:

1. *Orthogonality.* It is possible and desirable to treat coordination as orthogonal to computation for purposes of building programs.
2. *Generality.* It is possible and desirable to define *coordination* in such a way that it applies to *every* asynchronous software ensemble, from massively parallel, fine-grained applications through coarser-grained parallelism through dis-

Our article "Linda in Context" appeared in the April 1989 issue of *Communications* [8, p. 444]. According to comments by Kenneth Kahn and Mark Miller [27], the article generated a "flurry of electronic discussions." This article represents a full response to Kahn and Miller's thought-provoking critique.

tributed, heterogeneous and time-coordinated systems.

The concept of a coordination language follows from these claims. We introduced this term to designate the linguistic embodiment of a coordination model. The issue is not mere nomenclature. Our intention is to identify Linda and systems in its class as complete languages *in their own rights*, not mere extensions to some existing base language. Complete language means a complete coordination language of course, the embodiment of a comprehensive coordination model.

C is a complete computation language, though it lacks intrinsic support for process creation and interprocess communication. Linda is a complete coordination language, although it offers no support for arbitrary computations. We might call C an extension to Linda as reasonably as Linda is called an extension to C. But neither description is accurate, because Linda and C are, in concept, unrelated and orthogonal.

Orthogonality

What is a coordination language for? Is it not true that a computation language is useful by itself, while a coordination language can only be used in combination with a computation language? And does this not compromise our claim that the two are conceptually orthogonal?

No: a computation language by itself is useless. A computation must *communicate* with its environment or it serves no purpose. And the environment, insofar as it must ultimately be a person or people, can only be an *active* agent. A computation must perform operations whose purpose is to get information from or convey it to the environment, and these operations *are* a coordination language—often of a highly restricted, specialized or *ad hoc* sort, but a coordination language nonetheless. In the most primitive case, a program might spin while a user manually keys a bit string into a register. The coordination language in this case consists of operations like set the front-panel switches, read the accumulator, spin awaiting keyboard-buffer full. Not much; but these operations do provide a mechanism whereby two separate, asynchronous activities—the program and the person—can communicate.

Any computation language includes a sort of degenerate coordination language in the form of global variables and argument-passing: the separate parts of a program communicate with each other by using these mechanisms. But coordination is not merely information-exchange; the essence of our definition (and of the word's intuitive English meaning) involves information exchange among active agents. A coordination language must allow one active agent to convey information to another whose state is evolving and unpredictable. Assignment and parameter passing are insufficient.

In a broader setting, an operating system *defines* a coordination language for the benefit of the computations it supports. Computations need operating systems because they need to be tethered to other asynchronous activities and to create new activities. The lifelines that tie a computation to the outside world are (by definition) the province of a coordination language. Computations need to be tethered to users outside the machine (the operating system provides I/O). They need to be tethered to other activities, distant in time as well as (or instead of) in space—activities that preceded or will follow them. The operating system supports files. They need to be tethered to other computations across a network. They need to synchronize their activities (thus, implicitly, to communicate) with other computations that share the same machine. By providing a process model, and allocating system resources in such a way that activities can exist, the operating system carries out the basic "create activity"

Ensembles of all sorts dominate the near-term future of computing. Figuring out how to build and understand them will become the central problem of systems research.

role of a coordination language.

Operating systems tend to provide these functions in messy, *ad hoc* ways. Communication through time via the file system does not in the least resemble (so far as the syntax and semantics of operations offered to the programmer are concerned) communication with another process via shared table and semaphore, or communication across a network via message. And the coordination language defined by the operating system is likely to be strictly an *interpreted* language, not a compiled language. We do not generally feed programs into a "computation compiler" and also, separately, an "operating system" compiler, which would generate customized code for all external interactions. We rely on standard prepackaged libraries instead. But despite all this nonuniformity and *ad hoc*ness, the operating system exists to create activities, and to support their coordination with other activities: hence its primary function is to implement a coordination language. The major part of any operating system might be thrown out and replaced by an integrated, general-purpose coordination language.

In fact, this view of operating systems might induce some intellectual coherence in an important field that lacks all vestiges of it at present.

In sum, we cannot do anything useful with a coordination language standing alone. However, we cannot do anything useful with a mere computation language either. *All* useful computing depends on a combination of the two.

Generality as a Consequence of Orthogonality

Generality—our second claim—is suggested by orthogonality. The fact that we can separate computation and coordination does not mean that, in principle, we might not choose to cover the coordination spectrum with a million separate languages instead of a single integrated one. But the general-purpose computing language is a recognized, useful idea. It seems reasonable to posit a general-purpose coordination language as well.

We have staked our claims, and we turn now to defending them.

Why Separation? Why Generality?

Aside from Linda, few other systems accept the idea of a separate, conceptually self-contained coordination language. But, in effect, most current approaches reject separation.

If I accept separation, I might nonetheless reject generality: I might provide a coordination language designed for parallel applications exclusively, or distributed systems, or whatever.

Generality is almost *universally* rejected. The parallel-applications and the distributed-systems communities are (by now) almost completely disjoint. And as a rule, neither community finds much interest in the broader coordination issues discussed earlier—time-wise coordination, heterogeneity, software-human communication. There are pragmatic reasons why this should be so; but there are strong logical reasons (and some pragmatic ones as well) why it should not.

Opposing Separation and Generality . . .

Kahn and Miller ably state the case for integration:

> Concurrent logic programming (CLP) has traditionally been addressing another problem [as distinct from the problems Linda addresses]—namely how can one design a single language which is expressive, simple, clean and efficient for general purpose parallel computing?
>
> There is just one way to communicate in CLP, not as in Linda where there is a dialect specific way of communicating in the small and the tuple space way of communicating at the next level [27].

In other words, why worry about two separate toolboxes (a computation and a coordination box) when you can have *one*? A fair question. (Of course, from our point of view, *communication* is something that involves separate activities; a single process does not communicate with itself. Routines within one process pass information back and forth, but without confronting the problems of coordination among asynchronous activities. The "intra-process communication" that Kahn and Miller implicitly assume is communication in the sense that tossing a ball up and down is juggling. Nonetheless, this is a matter of definition, and they are obviously entitled to their own.)

We do not know of any equally concise attack on generality in our sense. So we will supply one, an argument that (we believe) many researchers would accept.

True, in some ultimate logical sense "communication is communication," but the pragmatic needs of (say) the distributed-system builder are very different from the needs of a parallel-applications developer. Accordingly, it is natural that completely different models have been developed to meet those needs.

For example, distributed systems often rely on remote procedure call. RPC is, in fact, a near-standard in this domain. For the builder of parallel applications, on the other hand, RPC is an unmitigated disaster. It is fundamentally wrong in concept: parallel programmers want to keep processes busy, want them to generate data and then get rid of it as quickly as possible. A communication model based on sending parameters to some routine, then awaiting a reply (while twiddling your thumbs) is rarely useful in parallel programming—to the extent that it *is* heavily used in some code, those are strong grounds for suspicion.

The rest of this article presents arguments in favor of separation

and generality, or (in other words) of the idea of a general-purpose coordination language.

These arguments are:

- In favor of separation: *portability* in a broad sense; and support for *heterogeneity*.
- In favor of generality: *economy*, *flexibility* and *intellectual focus*.

Separation: Portability and Heterogeneity

Portability means reusability, or recycle-ability in a broad sense. We would like to recycle applications, implementations, programming tools and (maybe above all) programmer expertise to the fullest extent possible. When moving from one platform to a different one, *or* from one parallelism model to a different one, *or* from one computing language to a different one, we would like to retain as much as possible. An integrated language sacrifices computing-language portability completely, and in many cases compromises the other varieties.

Given some C programmers, Scheme programmers and Prolog programmers, all of whom need to develop parallel applications, we could recommend three independent, tailor-made parallel variants of these languages—for example Concurrent C [16], Multilisp [19] and Parlog [26]. Alternatively, we could note that the machinery required for explicit parallelism is always the same, no matter what the base language: to get parallelism, we must be able to create and coordinate simultaneous execution threads. Given this observation, we can outfit all three groups in essentially the same way: we supply them with C-Linda, Schema-Linda and Prolog-Linda. In so doing we make it easier for them to switch base languages, simplify the job of teaching parallelism, and allow implementation and tool-building investment to be focused on a single coordination model.

The underlying premise is perhaps even more important: an XYZ programmer who needs to develop parallel applications will be supplied *not* with a new language, but with a dialect that is as much like XYZ as possible.

Heterogeneity is a generalization of portability. If our system works on X *or* Y, then it may very well work for X *and* Y. Certainly it is a better candidate for linking X and Y than some other model that is X- or Y-specific. Because a coordination language is not committed to any base computing language, it can work in principle with all of them, and tie programs in many languages together. Likewise with respect to mixed-machine or mixed-model heterogeneity.

Parallelism, Specifically

Before we shift focus to coordination in general, we need to consider how these arguments apply specifically to the domain that served as Kahn and Miller's main focus, and has been ours as well: models, tools and methods for parallel applications programming.

There are two radically different ways to support parallelism.

Approach I:

(a) Define all new languages for parallel programming (CSP was one of the first and remains one of the most influential); or

(b) generalize the semantic model of some base language to produce a new and complete parallel language (as in Multilisp [19], the concurrent logic languages, Concurrent Smalltalk [10] and many others); or

(c) sidestep the whole issue, by using sophisticated compiler or runtime technology to achieve parallel execution of programs that lack explicit parallelism. This approach includes work on parallelism compilers and on parallel execution of conventional functional or logic languages.

What these approaches share is the lack of any coordination model

Aside from Linda, few other systems accept the idea of a separate, conceptually self-contained coordination language. But, in effect, most current approaches reject separation.

per se. Instead, they supply a single, unified programming model (which may include the tools necessary to achieve coordination as one part of an integrated approach). The other approach to parallelism is:

Approach II:
Define an independent coordination model; this model can be added to any base language with no change to the base language semantics.

Linda falls into the Approach II category. Dongarra and Sorenson's Schedule [11] is another example. (Schedule, in turn, is related to Babb's work on coarse-grain dataflow [4].) Strand is an example that approaches the problem from a logic-programming viewpoint [15]. Linda is the only advocate of generality in this group: the others focus on parallelism specifically (although concurrent logic programming has a consistent secondary interest in distributed systems as well). This is not a criticism—merely a matter of differing foci.

Kahn and Miller's comments amount (fundamentally) to a criticism of Approach II from the standpoint of Approach I.

Under the first approach, parallelism is regarded as a generalization of some base language's computing model (if it is regarded at all). In the second—particularly Linda's version of the second—parallelism is a specialization of a more general phenomenon, the problem of coordination in all its guises.

There is no clear right and wrong between these two; both approaches have been used successfully. How do we evaluate them?

First, consider Approach I*a* and *b*. Kahn and Miller argue that these approaches offer one toolbox instead of two. We do not denigrate this argument from conceptual economy. Such an argument is exceptionally important. It should be overridden only in return for intellectual leverage of a decisive kind. We believe that portability, reusability and heterogeneity constitute this kind of overriding advantage. We also believe that, by sacrificing conceptual economy in the small, we will regain it in the large, when we unify the coordination tools required for parallel programming with a broad range of others. These arguments leave our respect for the basic principle of Kahn and Miller's argument undiminished. Obviously, each programmer will decide the issue individually.

Next, consider Approach I*c* (implicit parallelism) versus Approach II. The starting point for much of this work, particularly on implicitly-parallel functional languages, was the contention that explicit parallelism would prove too difficult for programmers to manage. "The potential performance of this kind of architecture is enormous," Turner wrote in 1982, referring to parallel machines, "but how can they be programmed? An idea that can be dismissed more or less straight away is that we should take some conventional sequential language and add facilities for explicitly creating and co-ordinating processes. This may work where the number of processes is small, but when we are talking about thousands and thousands of independent processes, this cannot possibly be under the conscious control of the programmer" [28 p. 10]. Although the field is still immature, the evidence to date suggests strongly that this contention is false, or at any rate misleading. Significant numbers of parallel machines have been installed and see routine use by programmers who use explicitly parallel methods. Currently these architectures are more likely to involve tens or hundreds than thousands of processors, but why existing techniques should suddenly fail at the transition from (let us say) a 7-Cube to a 10-Cube is unclear.

The fallacy in Turner's statement has proven to be the underlying assumption that somehow each process in an ensemble will be created separately and treated as an individual. In the context of large-scale Linda programs, which usually involve many identical worker processes, or processes each of which computes one piece of a large, aggregate data structure (turning into the result upon completion), Turner's statement makes no more sense than the claim that "DO loops may work for small numbers of iterations, but when we are talking about thousands and thousands of iterations, this cannot possibly be under the conscious control of the programmer . . . " Of course it can't, but so what? To specify explicitly does not mean to specify *individually*.

In evaluating Approach I as a whole, there is a final pragmatic factor to keep in mind as well. Those who work actively at the interface between computer science and real computation are aware of the fact that most "real" parallel applications begin life *not* as blank sheets of paper, but as serial programs that run too slowly. If we were asked for help in parallelizing a large serial code, and we opened the discussion from our side by saying "first thing, throw out every line of this program and rewrite it in (say) Miranda," the response would likely be unprintable. Not because real-world programmers are too stupid to understand the beauties of these brave new languages. It is merely that they have better things to do with their time than rewrite code that already works, unless the advantages of doing so are overwhelming. They rarely are.

Of course many parallel applications *are* built from scratch, and their numbers will increase. But forcing new languages on parallel programmers clearly complicates the transition to parallelism.

Generality: Economy, Flexibility and Intellectual Focus

We turn now to basic claim number two: in principle, you can use the *same* coordination language that you rely on for parallel applications

programming when you develop distributed systems. You can use the same model in building (or at any rate conceptualizing—designing the programmer interface to) a file system. You can use the same model, again, in building heterogeneous applications. And you can use the same model for implementing basic human-machine communication. This approach does *not* accord with current practice, to say the least. Before we consider why you might want to do this, we need to consider whether (in basic design terms) the whole thing is even possible.

We are arguing on behalf of a class of systems, not Linda specifically. But it is convenient to use Linda as an example. All these forms of communication are subsumed, in logical terms, by the Linda model. They are not all *supported* adequately by current implimentations; but that's another question. The first question is: can you *design* a general-purpose coordination language?

The Linda model has been described often, and we will not repeat the description here. However, in brief outline, Linda provides an associative object memory, conceived as a kind of stretchy envelope. Processes in Linda inhabit this envelope, called a *tuple space*. When they have information to communicate, they generate tuple-structured data objects and release them into the envelope. When they need information, they may *read* a data object or *consume* one, as the context requires. Objects are described for purposes of reading or consumption by an associative naming scheme that operates like "select" in a relational database. Processes turn into tuple-structured data objects, indistinguishable from all the rest, when they are done computing. The system as described is supported commercially on a broad range of platforms. Current research implementations (epitomized by Jagannathan's Schema Linda [20]) support multiple first-class tuple spaces: whole tuple spaces can appear as tuple fields; whole tuple spaces may be manipulated as unit objects.

The preceding is brief and sketchy, but it is sufficient to motivate our claim that Linda can support all forms of communication listed earlier. Inter-process communication, of the sort that parallel applications and distributed systems both require, is realized in terms of distributed data structures. Information-producing processes build data structures out of tuples; information-consuming processes read or consume those structures. (In the simplest case, such a structure is merely a single tuple.) The technique is discussed at length in [8, 9].

RPC is trivial to simulate: the "invoke procedure" operation is implemented by a "generate parameters" object followed by a "consume result" operation. The remotely invoked procedure becomes a process which repeatedly accepts a parameter object, invokes the called-for procedure locally and then generates a result object. (The converse, by the way, is not true. Linda operations cannot be translated directly into RPCs—which procedure would they invoke? Not some hypothetical object store procedure, because runtime efficiency is mandatory for parallel applications, and we cannot allow centralized bottlenecks. Nor can we allow Linda's asynchronous, nonblocking "generate object" operation to block until a remote procedure generates a logically pointless reply. Nor can we readily support Linda's view of processes as incipient data objects in this framework.)

Linda's boundaries as a parallel applications tool continue to expand. For example, one recent Linda application in financial analysis successfully uses the owner-computes model of data parallelism. Jagannathan and Philbin's STING system supports fine-grained Linda programs, currently on shared-memory multiprocessors but with a port to distributed-memory environments planned. It promises to expand significantly the range of program structures that can be expressed cleanly and implemented efficiently using the Linda coordination model [21]. Scientific Computing Associates now supports Fortran-Linda as well as C-Linda. The "Piranha" system, which executes Linda programs on conventional local networks in such a way that idle workstations may join an ongoing computation, and participating workstations may withdraw quickly when their owners need them, runs on 60 workstations in the Yale Computer Science Department and has been used to execute a variety of production applications. At several recent workshops (notably *Research directions in high-level parallel languages*, which focussed on Unity, Gamma and Linda [24], and *Linda-Like Systems and Their Implementations* [23]), Linda-related projects addressed to a broad range of other programming styles was presented.

Linda is inherently a (distributed) file-system and database model as well, because tuples are persistent objects. A tuple space is a sort of file: objects can be added to and read from the file; tuples are immutable, but they are modified in effect by removing an old one and reinstating an updated version. Associative addressing makes it possible to organize the tuples in a "file" as an indexed stream of bytes (one byte per tuple); such files can also hold a heterogeneous stream of arbitrary records, or an unordered collection of objects. They can hold processes (incipient tuples) as well as data objects. Thus we might, for example, store a librarian daemon inside a mail file, and so on. On the use of Linda in database setting, see Anderson and Shasha's work on "Persistent Linda," which supports transactions and some other extensions that are useful in this domain [3].

Linda is logically suited to language-heterogeneous applications: the Linda coordination model makes no reference to any particu-

lar host computing language. Obviously, this statement begs the hard question of type compatibility—each language puts data objects created according to its own type system in the tuples it generates. But *if* we define some common (stripped-down) type system, the Linda operations make it possible for processes in many languages to collaborate on production and consumption of a single shared data structure, make it possible for a process in one language to consume the residue (the tuple left behind upon completion of the computation) of a process expressed in another language, and so on. Linda supports basic man-machine communication insofar as a user, using a Linda command interpreter, can dump objects into tuple space, read or retrieve them directly.

The fact that all this holds in concept does not mean, of course, that it holds in practice. For example: the RPC community has invested considerable effort in defining the meaning of its construct in the presence of various failures, and in developing implementations that are robust in the face of network faults. The fragmentary coordination language defined by the typical file system is supported by an implementation optimized to the needs of I/O interfaces, to security and authentication requirements, and so on. These issues are *not* confronted by current Linda implementations which target parallel applications where runtime performance (not reliability, security and so on) is the driving consideration.

Still, it is vital not to loose sight of the underlying question. In this section we ask not *what has been implemented* but what *might be and ought to be implemented.* The fact that current RPC implementations are well-suited to the pragmatics of distributed systems is, for now, formidable and important. But obviously, the same sort of research effort that led to a reliability semantics for RPC could (and *will*, we expect) lead to the same sort of thing in the Linda context, or in the framework of some other general-purpose coordination language. A number of interesting projects have already addressed important aspects of the reliable Linda problem / for example, [5] and [23]). (Given our own steadily-increasing focus on local networks as parallel machines [1], we will also be confronting many of these issues.) Likewise for issues of file systems, databases and so on.

In short, a general-purpose coordination language *is* possible in concept. But is it a good idea?

Generality: Economy and Flexibility

Conceptual economy is a principle of great importance: Kahn and Miller made this argument (as have many others in recent millennia), and we accepted it. We would rather have a single coordination toolbox than many separate ones.

The practical gain from conceptual economy is *flexibility.* Simple, economical languages tend to be supple and powerful, complex ones tend to be rigidly inflexible—a stubborn fact that emerges screaming from programming language history, only to be repeatedly ignored. (Our recent textbook on programming language design [18] discusses this issue at length.) By proposing a single, general-purpose coordination language we are filling in the blanks that separate massive parallelism from task-level parallelism or distributed systems—leaving ourselves with a clean and continuous spectrum stretching from one end of the coordination world to the other. Two possibilities—interpolation and extrapolation—follow. We get strong support and a conceptual basis for applications that do not fit precisely into any one category. Also, we can smoothly extend our knowledge of coordination *outward* beyond the software world altogether, into the domain of ensembles in general—including, for example, human ones.

Interpolation

Consider two active research projects in our group, one dealing with realtime data fusion (the trellis [12, 14]), the other with expert databases (the FGP machine [13, 17]).

The trellis is a software architecture that uses parallelism for clarity, insofar as parallelism allows us to impose a uniform framework on a wide-ranging collection of separate programs, and speed, insofar as parallelism allows us to guarantee sufficient execution resources to meet realtime deadlines. But the project does not stop with the trellis program itself. Factor et al. [14] describe the front-end visualizer running on a graphics workstation. In the complete system we envision, Linda supports parallelism in the trellis which runs on a parallel machine. It supports uniprocessor concurrency within the graphics workstation (processes that manage subsidiary windows require data from the main display-manager process). It also supports distributed system communication between the workstation and the trellis. The first two parts of this picture, parallelism and uniprocessor concurrency, are complete, and the third is current research.

Question: Why should we accept *three* toolboxes, one for parallel applications (say, message passing), one for uniprocessor concurrency (for example, shared memory with locks), and one for trans-network communication (say, RPC), when *logically* Linda works well in all three cases? And what do we call this program, anyway? A parallel application? A distributed system? Clearly it's an *ensemble* pure and simple.

The FGP project poses a similar question. Its database manipulation component is computationally expensive for large databases, and is now being parallelized. But the full system goes beyond a single instance of the program: the goal is to support the expert examination of a local database running on a local workstation or PC, and a simultaneous examination of a much larger (public) database to be executed on a parallel machine. In a medical

domain, for example, the clinician runs a local search against his own patient database while simultaneously searching a hospital's much larger case repository. Some of the requisite communication has to do with parallelism and some with distributed systems, but logically it is all the same. Why should we use two separate communication systems when we only *need* one?

These projects refuse to be neatly categorized; they squirm wherever you put them. They clearly require coordination tools that are powerful enough to work in many settings. And there is nothing unique about our research effort in this regard. Mixed-mode ensembles will be normal and widespread in the future.

Extrapolation

Once we have identified coordination as a topic that we can discuss in general, we can consider applying our knowledge of software ensembles to the construction of other kinds of ensembles—human ones, for example. (The questions we face in this new area are similar to the ones Thomas Malone poses in his provocative work on coordination theory [25].)

For example: communication in Linda is based on *distributed data structures,* or shared structures such as streams and arrays that are built out of tuples. Processes communicate via these shared structures. But in principle, people might also communicate this way. We might imagine a tuple space surrounded by people who release, read and retrieve tuples directly. Alternatively, each person might be represented by a software agent—a process inside of tuple space—that is active on his behalf.

We can now build information-sharing software in which each tuple is an information object. Many diverse, sometimes complex questions and requirements have simple solutions in terms of distributed data structures.

Another example exists in multiple tuple space Linda systems. Multiple tuple spaces are useful in structuring software, but they are also a natural mechanism for building a hierarchical "conceptual landscape" to describe a project or organization. When users want to know something about Linda itself, for example, we might refer them to a nest of tuple spaces that captures the system's conceptual structure. Within the global Linda space are tuple spaces holding documentation, code, reports, or perhaps running programs. Each of these contains appropriate subspaces in turn. We arrive at a structure that resembles a hierarchical file system, except that the objects being organized are full-fledged tuple spaces. They may contain processes (daemons or visiting agents) as well as data objects.

A final example: in [8] we use the term "Turingware" to refer to an ensemble incorporating people *and* processes, in such a way that no element knows or cares whether the others with which it deals are processes or people. One current project involves a Turingware version of the trellis architecture discussed earlier.

In short: when we introduce general-purpose coordination models, the resultant broadening of intellectual scope is wide-ranging and considerable.

Intellectual Focus

The idea of a general-purpose coordination model directs attention to the fact that there *is* such a topic as ensemble building in general, and that it is a fundamental issue for computer science.

Programming languages have traditionally treated I/O, the file system and the relationship between a user's program and the surrounding environment as outside the bounds of a computing model, an area for recourse to extra-linguistic library routines or *ad hoc* extensions. (Of early languages, Cobol and APL were each partial exceptions in different ways, but neither was influential in this respect.) Thus Algol 60, for exam-

A general-purpose coordination language *is* possible in concept. But is it a good idea?

ple, has no provision for I/O; it assumes that I/O will be handled by hand-coded external routines.

Consider a *Gedankenexperiment* based on an anti-Algol: this language makes no provision for computing values; it assumes that values will be computed by external library routines. It is precisely a coordination language, capable of expressing interactions between running programs and users, the generation, storage and retrieval of persistent objects in a file system, and the coordination of multiple activities into a single ensemble.

If 1960 had seen the definition of anti-Algol instead of Algol, we might have developed a set of value-computing tools as unsystematic and *ad hoc* as our present coordination tools. Of course, this scenario was impossible, not only on obvious pragmatic grounds but because of the existence in recursive function theory of a simple and comprehensive model of computation. But the experiment should give us at least a moment's pause, because the current direction of computing makes it appear that anti-Algol and not Algol might ultimately prove the more important language.

In an age where prepackaged software is cheap and for sale everywhere (no doubt there will be software vending machines before long), the programmer's main task will shift decisively in the direction of *gluing components together*—building ensembles. (Modern IC's have shifted the digital designer's role in the same way, towards the gluing-together and away from the synthesis of components.)

Furthermore, a general coordination model is the basic mental construct that we require in order to distinguish programming from mathematics. Every computation language is a fancy Turing Machine. But programmers do not deal in the mere evaluation of expressions, precisely because asynchronous ensembles are the fundamental fact of programming. Even when we deal with a conventional, deterministic, single-threaded application, the user plus the computer constitutes a two-part asynchronous ensemble. An ensemble is the natural outgrowth of the inevitable asynchronism in any human-computer system. Why not three or n activities instead of two?

It would be nice to have a theoretical foundation for general coordination. We would like to see the following characteristics in such a model. First, a simple definition of computational space and time where a point in space is identified with a single locus of control (or a single Turing Machine), and a point in time is defined as the current states of many loci or TMs. Second, a model that becomes a TM when projected onto the time axis at some spatial point, and becomes a "current coordination state" when projected onto the space axis at some temporal point. From our standpoint, a "current coordination state" is (the current, frozen state of) a tuple space: a TM and a tuple space are orthogonal elements in computational time-space.

Conclusions

A broad research effort aimed at the development of general-purpose coordination languages is long overdue. The tangible result would be a tool of great power and significance. The intangible one would be a better understanding of the root problems of computer science. There appears to be an unspoken consensus in much of the research community that every twist and turn in the hardware development path, particularly where parallel machines or networks are concerned, calls for a new language or programming model, a new design, new implementation and new coding methods. In the long run, this approach is intellectually crippling. What are the *fundamental questions* here?

Although we have used the Kahn and Miller comments as a foil for this exposition of our basic premises, our work is, in fact, closely allied to theirs. The issues they raise in their "Open Systems" paper [22] are important, and the dynamic, evolving and open-ended systems they envision will become increasingly central to systems research. Our thinking about Linda and its evolution has been strongly influenced by their work.

Linda obviously shares much with other coordination-language projects, particularly with Dongarra and Sorenson's Schedule (at any rate with respect to basic underpinnings). We also see strong similarities between our approach and the work of Bisiani, Forin and Ambriola on heterogeneity and coordination [6, 7]. In general we see computation and programming languages as areas in which further progress will be slow, incremental and, in many cases, of marginal importance to working programmers. Coordination languages are a field of potentially great significance. A growing number of groups will play major roles in this work.

Acknowledgements

The authors thank Paolo Ciancarini, Mark Day, Suresh Jagannathan, Steven Lucco, Thomas Malone and Ross Overbeek for illuminating comments about the ideas in this article.

References

1. Arango, M., Berndt, D., Carriero, N., Gelernter, D. and Gilmore, D. Adventures with network Linda. *Supercomput. Rev. 10,* 3 (Oct. 1990) 42–46.
2. Agha, G., *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press (1986).
3. Anderson, B. and Shasha, D. Persistent Linda: Linda + Transactions + Query Processing. In *Research Directions in High-Level Parallel Languages.* D. LeMetayer, ed. (Mont Saint-Michel: IRISA-INRIA, June 1991): Springer Verlag (forthcoming).
4. Babb, R.G. Parallel processing with large grain data flow techniques. *IEEE Comput. 17* (1984) 55–61.
5. Bakken, D.E. and Richard D. Schlichting, Tolerating Failures in the Bag-of-Tasks Programming Paradigm. In *Proc. of the 21st Int.*

Symp. Fault-Tolerant Computing, Montreal, Canada (June 1991), 248–255.

6. Bisiani, R. and Forin, A. Multilanguage parallel programming on heterogeneous machines. *IEEE Trans. Comp. 37*, 8 (Aug. 1988) 930–945.
7. Bisiani, R., Lecouat, F. and Ambriola, V. A tool to coordinate tools. *IEEE Software* (Nov. 1988) 17–25.
8. Carriero, N. and Gelernter, D. Linda in context, *Commun. ACM 32*, 4 (Apr. 1989) 444–458.
9. Carriero, N. and Gelernter, D. *How to Write Parallel Programs: A First Course*. MIT Press (1990).
10. Dally, W.J. Object-oriented concurrent programming in CST, in Proc. Third Conf. on Hypercube Concurrent Computers and Applications, (1988) p. 33.
11. Dongarra, J.J., Sorenson, D.C. and Brewer, P. Tools and Methodology for Programming Parallel Processors, in *Aspects of Computation on Asynchronous Processors*, M. Wright, Ed. (North Holland, 1988) pp. 125–138.
12. Factor, M. The Process Trellis Software Architecture for Parallel, Real-Time Monitors. Yale Univ. Dept. Comp. Sci., PhD. Dissertation (Oct. 1990)
13. Fertig, S. and Gelernter, D. A Software Architecture for Acquiring Knowledge from Cases. In *Proc. of the International Joint Conference on Artificial Intelligence*, Sidney, Australia, Aug. 1991.
14. Factor, M., Gelernter, D., Kolb, C., Miller, P. and Sittig, D. Real-Time Data Fusion in the ICU. *IEEE Computer*, Nov. 1991, 45–55.
15. Foster, I. and Taylor, S. *Strand: New Concepts in Parallel Programming*. Prentice-Hall (Englewood Cliffs, NJ, 1989).
16. Gehani, N. and Roone, W.D. *The Concurrent C Programming Language*. Silicon Press, 1989.
17. Gelernter, D. Multiple tuple spaces in Linda, in *PARLE '89*, E. Odjik, M. Rem and J.-C. Syre, Eds. Springer-Verlag: 1989, 20–27.
18. Gelernter, D. and Jagannathan, S. *Programming Linguistics*. MIS Press, 1990.
19. Halstead, R. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. and Sys.*, Oct. 1985.
20. Jagannathan, S. Optimizing Analysis for First-Class Tuple Spaces. In *Languages and Compilers for Parallel Computing II*. D. Gelernter, T. Gross, A. Nicolau and D. Padua, Eds. MIT Press/Pitman Publishing, 1991 (forthcoming).
21. Jagannathan, S. and Philbin, J. Preliminary STING benchmarks. NEC Institute Princeton Internal Memorandum, Dec. 1991.
22. Kahn, K.M. and Miller, M.S. Language Design and Open Systems, in *The Ecology of Computation*. B. Hubermann, Ed., (North Holland, 1988) 291–314.
23. Kambhatla, S. and Walpole, J. Recovery with limited replay: Fault-tolerant processes in Linda. Oregon Grad. Inst. Dept. CSE TR CS/E 90-019 (Sept. 1990).
24. LeMetayer, D., ed. *Research Directions in High-Level Parallel Languages (Mont Saint-Michel: IRISA-INRIA, June 1991): Springer Verlag (forthcoming)*.
25 Malone, T.W. What is coordination theory? MIT Center for Info. Systems Res. Working Paper 182 (Feb. 1988).
26. Ringwood, G.A. Parlog86 and the dining logicians, *Commun. ACM 31*, 1 (Jan. 1988) 10–25.
27. Technical correspondence. Linda in context. *Commun. ACM 32*, 10 (Oct. 1989) 1244–1258.
28. Turner, D.A. Recursion Equations as a Programming Language, in *Functional Programming and its Applications*. J. Darlington, P. Henderson and D.A. Turner, Eds. Cambridge University Press (1982) pp. 1–28.
29. Wilson, G. ed., *Linda-Like Systems and Their Implementations* (Edinburgh Parallel Computation Centre, June 1991).

CR Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Language Classifications—*parallel languages;* D.3.3: Language Constructs

General Terms: Coordination, Ensembles, Parallelism, Languages

Additional Key Words and Phrases: Coordination languages, Linda

About the Authors:

NICHOLAS CARRIERO is an associate research scientist in the Department of Computer Science at Yale University and a research scientist at Scientific Computing Associates, New Haven, Conn. His research interests include parallelism, compiler techniques and programming languages.

DAVID GELERNTER is an associate professor of computer science at Yale University. His research interests include parallelism, programming languages and artificial intelligence.

Authors' Present Address: Department of Computer Science, Yale University, New Haven, CT 06520, carriero@cs.yale.edu, gelernter@cs.yale. edu.

This work is supported by National Science Foundation Grant CCR-8657615 and by the Air Force Office of Scientific Research Grant AFOSR-91-0098.