A Program Building Tool for Parallel Applications*

Shakil Ahmed, Nicholas Carriero and David Gelernter
Department of Computer Science
Yale University

December 1, 1993

Abstract

We describe the Linda Program Builder — a higher-level programming environment that supports the design and development of parallel software. It isolates much of the administrative effort in constructing parallel programs, and maintains a program-describing database. This database feeds information to the compiler for optimization, to a visualizer for enhanced program visualization, and potentially to other tools in the environment. The LPB is a window-oriented, menu-based, user-friendly system which provides coordination frameworks for program construction. These templates or coordination frameworks themselves can be custom-constructed by invoking a template-building template; most importantly, the LPB represents the idea of an "open" or "dynamic" preprocessor as an alternative to new programming languages .

*This research is supported by National Science Foundation grant CCR-8657615, by the Air Force Office of Scientific Research under grant number AFOSR-91-0098, and by Scientific Computing Associates, New Haven. Linda is a registered trademark of Scientific Computing Associates, New Haven, CT.

1 Introduction

The Linda Program Builder (LPB) [1] [2] is a higher-level programming environment that aids in the design and development of parallel software. The LPB is an Epoch-based[1], menu-driven, user-friendly system that supports incremental development of explicitly parallel C-Linda programs.

This paper will discuss three major topics: program construction with the LPB, the LPB's interaction with the compiler and visualizer, and the idea of an "open" or "dynamic" preprocessor like the LPB as an alternative to new programming languages.

Linda itself has been extensively discussed in previous work, so we will not describe it here. [9] is a representative paper. Two important points respecting the LPB's relationship to Linda are important, though. (1) Nothing about the "parallel program builder" approach restricts it to Linda. What we mean by "program builder" is a tool which captures idiomatic use of a particular language, and guides the user through program development, by offering higher level constructs that are transformed into some base language. The sort of tool we describe might be built in the context of any explicitly parallel programming environment. On the other hand, (2) the choice of Linda isn't arbitrary either.

2 Program Construction with the LPB

The LPB is built on top of Epoch running under X-windows. The LPB environment is menu-driven, but allows the full flexibility of Epoch (Emacs) in editing all files. Programs are constructed interactively. The LPB can also be used to parallelize an existing program by employing its commands to insert parallelization code. Several windows are open at all times, offering command menus and information on the developing program's current status. The template menu offers a selection of available templates. The tuple menu presents a choice of currently-known tuples. The operations we may perform on these tuples are listed in the command menu which changes depending on which tuple is currently selected in the tuple menu.

An experienced programmer may choose to bypass many of the point-and-click facilities of the LPB. There have been many template-editor predecessors to the

[1]Epoch is a multi-window version of emacs developed by S. Kaplan of the University of Illinois, Urbana

LPB, notably the Cornell Program Synthesizer[27], but on the whole, they impose rigid frameworks which the programmer is forced to follow. Requiring the programmer to follow an imposed template guarantees syntactic correctness, but may cramp a creative programmer's style by restricting his flexibility. The LPB offers similar features, but doesn't impose them.

The LPB captures organizing strategies for parallel programs and basic coordination frameworks. The basis for the "coordination framework" approach to parallel programming is presented in [7]. This method focuses on three main paradigms, called "specialist", "result", and "agenda" parallelism. The most important features of the LPB are its support of *templates* and *high-level operations,* and its construction of a *program database.* The latter is the medium through which the LPB interacts with other tools in the environment. Templates represent entire program structures and direct control flow. There is also support for manipulating distributed data structures. Finally, high-level operations provide abstractions to a user which get transformed into the base language before compilation.

2.1 Templates

A template is a program skeleton for a particular paradigm that serves as a blueprint for program construction. A template expands into more detailed code. A programmer can choose to follow a template all the way through, but he is free to leave this framework whenever he wants and to return when necessary. There is a tradeoff here between flexibility and allowing potential consistency problems. Should a programmer decide to bypass the LPB and code constructs that contradict its established patterns, an inconsistency between the program database and the program itself might be created. The LPB will warn the programmer when this happens, but cannot overrule what he has done.

Many templates have already been implemented, but others may be added by using a special template-building template. Templates are expanded incrementally using buttons. Consider the master-worker template as an example. The master-worker paradigm is discussed in [1], and we omit details here. A master process creates tasks. Worker processes operate in parallel, grabbing tasks as they become available and generating results for a task when they are done with it. The master process gathers these sub-results and determines the final result. When a master-worker template is chosen from the main menu, a template skeleton is presented. Figure 1 shows the initial buttons. Clicking a template button causes it to expand into more code, which may contain more buttons. Various stages may require further

Figure 1: Initial master-worker template

input, for which the user is prompted either by menus or input windows. The program grows as more buttons are expanded and more information is obtained from the user. The buttons may be ordered in priority, i.e. some buttons may be dependent on information that can only be acquired by first expanding other buttons.

A master-worker program typically involves task tuples and result tuples. All tuple operations and the associated variable declarations are automatically generated according to the particular model chosen. An intermediate stage in program generation is shown in figure 2.

The user is always free to enter code or comments anywhere within the partially constructed program.

There are several other templates for developing parallel programs, some supporting completely different methodologies. The data parallel template, for example, supports the owner-computes style of programming by providing filter, log and merge routines. A set of global reduction operators completes the data parallel package. These operators are implemented as macros, but could equally well be communicated to the compiler for custom optimizations. The LPB interface for data parallelism thus smoothes the way for Linda programmers who use the data parallel model.

2.2 Distributed data structures

Linda programs generally make use of distributed data structures such as distributed arrays, task bags, shared variables and so on. Many of these crop up within the coordination-framework templates discussed above. Structures such as task bags, watermarked bags and distributed queues, to name a few, are often incorporated into the choices presented during the construction of a program through a template. But if a programmer needs to specify a data structure outside of a specific template, the LPB still provides support.

For example, the LPB has menu options to support creating and manipulating shared variables and counters in tuple space. Counting semaphores are supported in a similar manner. The labels of these variables appear in the tuple menu. Picking a tuple of one of these types will cause the commands menu to change accordingly, and the information window to display known information on the tuple.

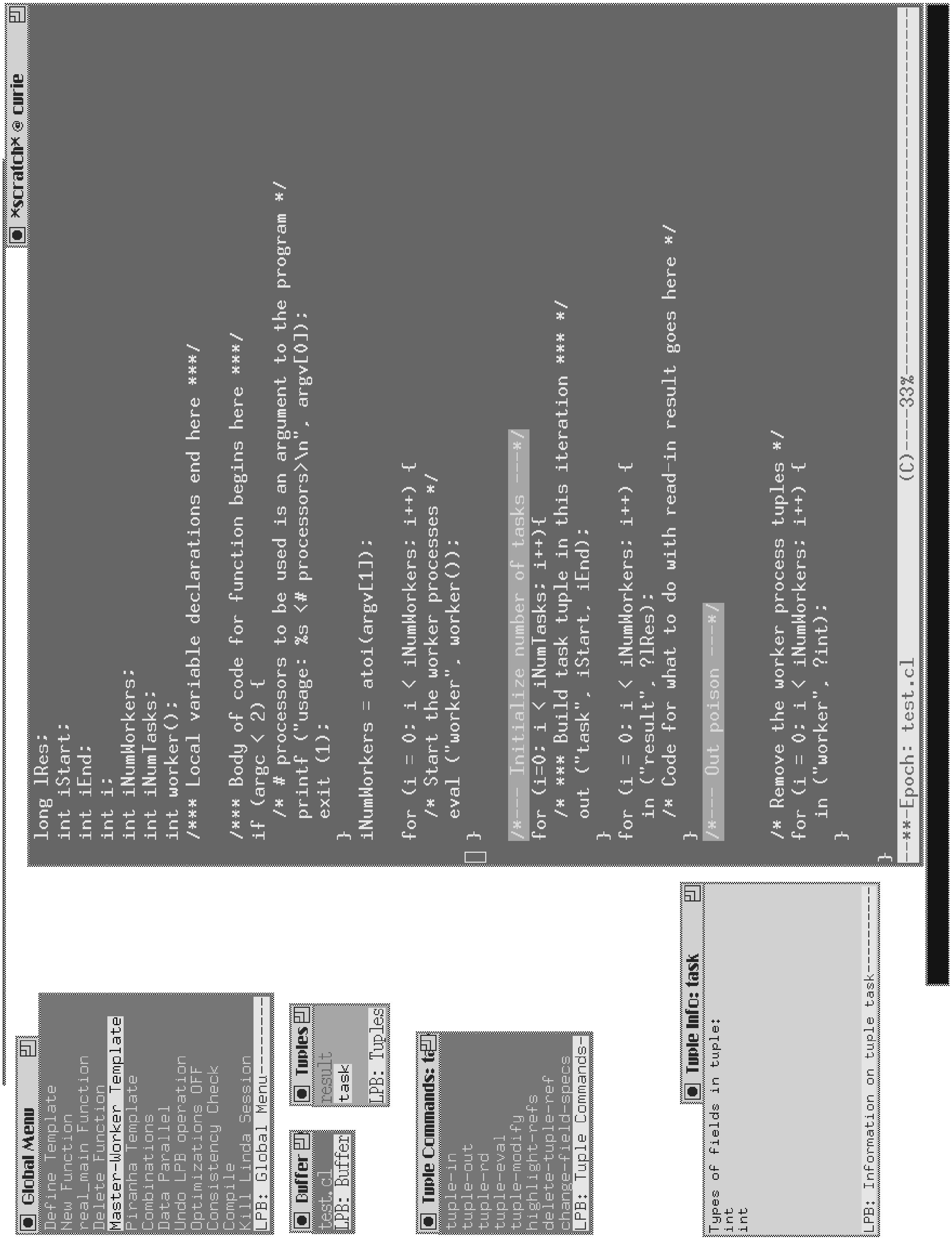

Figure 2: Intermediate stage in master-worker program

Distributed queues of various kinds are often required in parallel programs. They may have multiple sources, sinks, or both. The synchronization and handshaking necessary for coordination among the various sources and sinks can be achieved through distributed head and tail pointers in tuple space. The LPB provides a complete set of menu functions to create and manipulate queues. Upon selection of a "create-queue" command, a popup menu offers choices. Once a model has been selected, all the tuples necessary for maintenance of the queue are automatically generated and initialized. A user is now free to select menu commands to add to or remove from the queue as desired. All tuple operations, declarations, and support code are automatically inserted at the appropriate places.

2.3 Higher level program constructs and abstractions

Certain operations ought to be supported, but not at the level of a full program template which directs control flow. The LPB supports high-level operations which can be expanded and then abstracted back again (buttons, on the other hand, can only be expanded). Thus a higher level construct may be presented to the user as an abstraction. If the user wants to see the implementation of the higher-level operation, he can choose to expand it. It can then be abstracted back to the higher level representation, which is more concise and easier to understand.

The abstraction feature is a useful tool in constructing and viewing programs. It supports top-down programming by presenting a high-level view of program structure that can be expanded downwards at will, but also abstracted back up again to a conceptually more appealing higher-level format. This allows the programmer to concentrate on hierarchical program construction at a high level, and to deal with "blocks" of code represented by abstractions.

Abstractions in the LPB are somewhat similar to the Cedar [33] approach in its Tioga structured text-editor. Tioga treats documents in a tree-structured manner where each node is a paragraph or a statement. This hierarchical node structure allows detail to be concealed in the interests of a conceptually higher-level view, much as in the LPB. A related feature appears again in editors that provide outline modes, allowing for a hierarchical perspective on text. Note however that the abstraction feature of the LPB involves considerably more than mere in-place expansion. Expanding an abstraction involves actual code generation which may affect code spread across several modules. This code is automatically generated, and it is not a mere rehashing of something the user has explicitly declared previously.

As an example, we present the **or-in** construct, which allows the user to **in** a tuple **or** another one or another one and so on. The LPB implements the **or-in** function as a higher-level operation. If the user selects the menu option for an **or-in**, a menu pops up with a list of the tuples that are known to the database. The user selects from these the tuples that will appear within the **or-in**. The **or-in** appears to be a regular program construct, but the relevant lines in the code are highlighted (Figure 3).

The highlighting indicates that it is a higher-level operation. Expanding this abstraction will show the user how it is implemented in Linda code. The **or-in** becomes an **in** of a bit vector to check which tuples are available. Next is a conditional to check which bit is on; based on the check, the appropriate tuple is read. The bit vector has to be generated whenever one of the tuples of the **or-in** is used in an **out** or **eval**. Thus, the expansion causes all relevant references to those tuples in all open modules to be followed by a new **out** which generates a bit vector with the bit corresponding to the **out**'ed tuple turned on (Figure 4). If the cursor is placed on the main section of the **or-in** expansion, and the abstraction menu item is selected, all the expansion details disappear, and the abstraction reappears, making the **or-in** look very much as if it is a part of the language.

2.4 The program database

The LPB continuously updates a program-describing database. Every tuple, function, abstraction, higher level operation, or other significant component of the program is entered into the database as it is used. The database keeps information on a tuple's label, on the variables used in its fields, the status of each of the fields, information on the nature of the tuple and its use, and a record of all the places where references to the tuple exist. The database also keeps track of which buttons have been expanded, and of the general state of the template.

The archive is global across a user's LPB sessions. It is saved together with the program files, and automatically loaded when a file is read in.

This database is the backbone of the LPB, maintaining all the information necessary to implement higher level operations and provide user support. This information is used to eliminate memory-work and reduce keystrokes. For most tuple operations, for example, variable declarations and code insertion are automated, as are cross-module propagation of updates to tuple references when a tuple structure is modified. The database also supplies information to the compiler and visualizer.

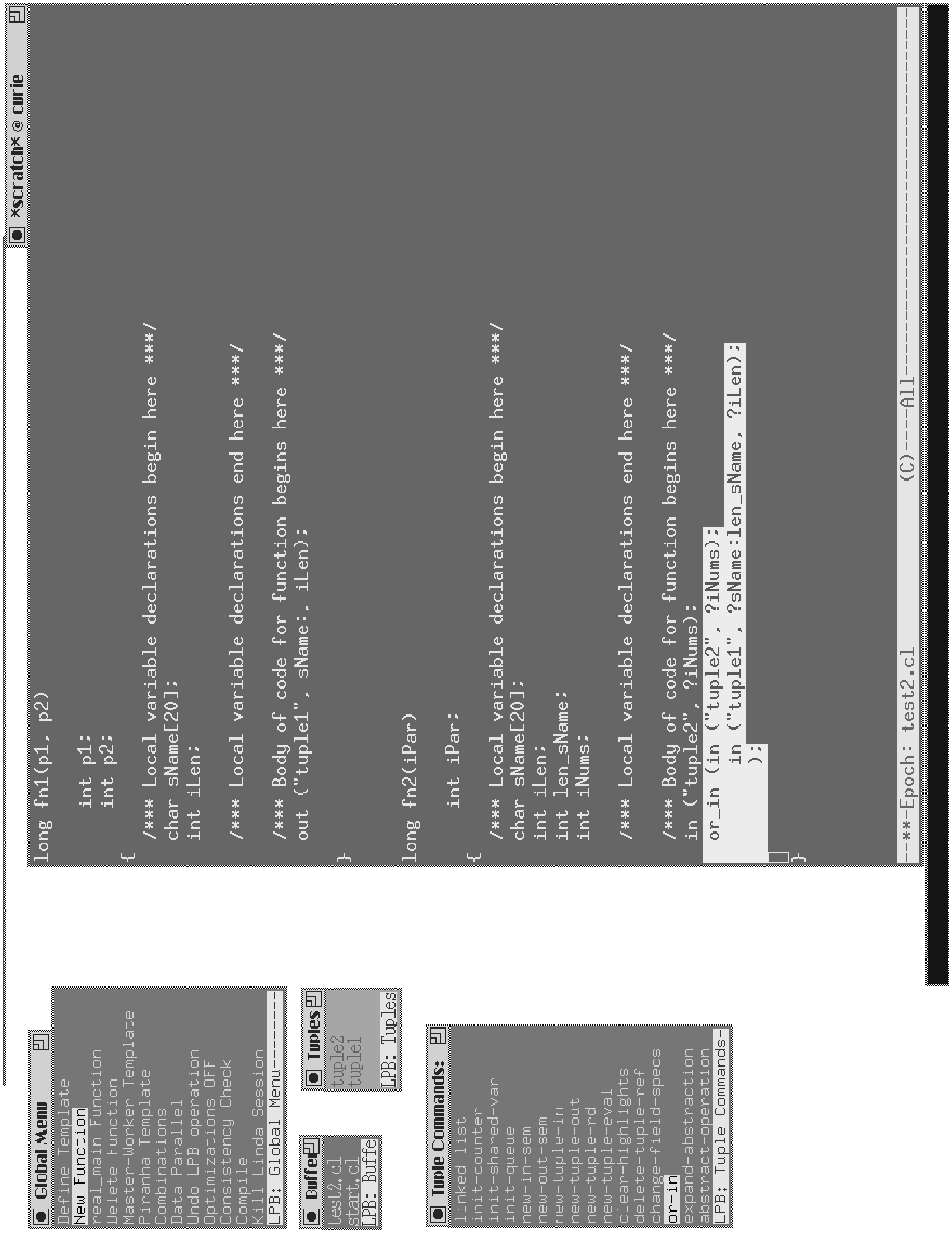

Figure 3: The **or-in** abstraction

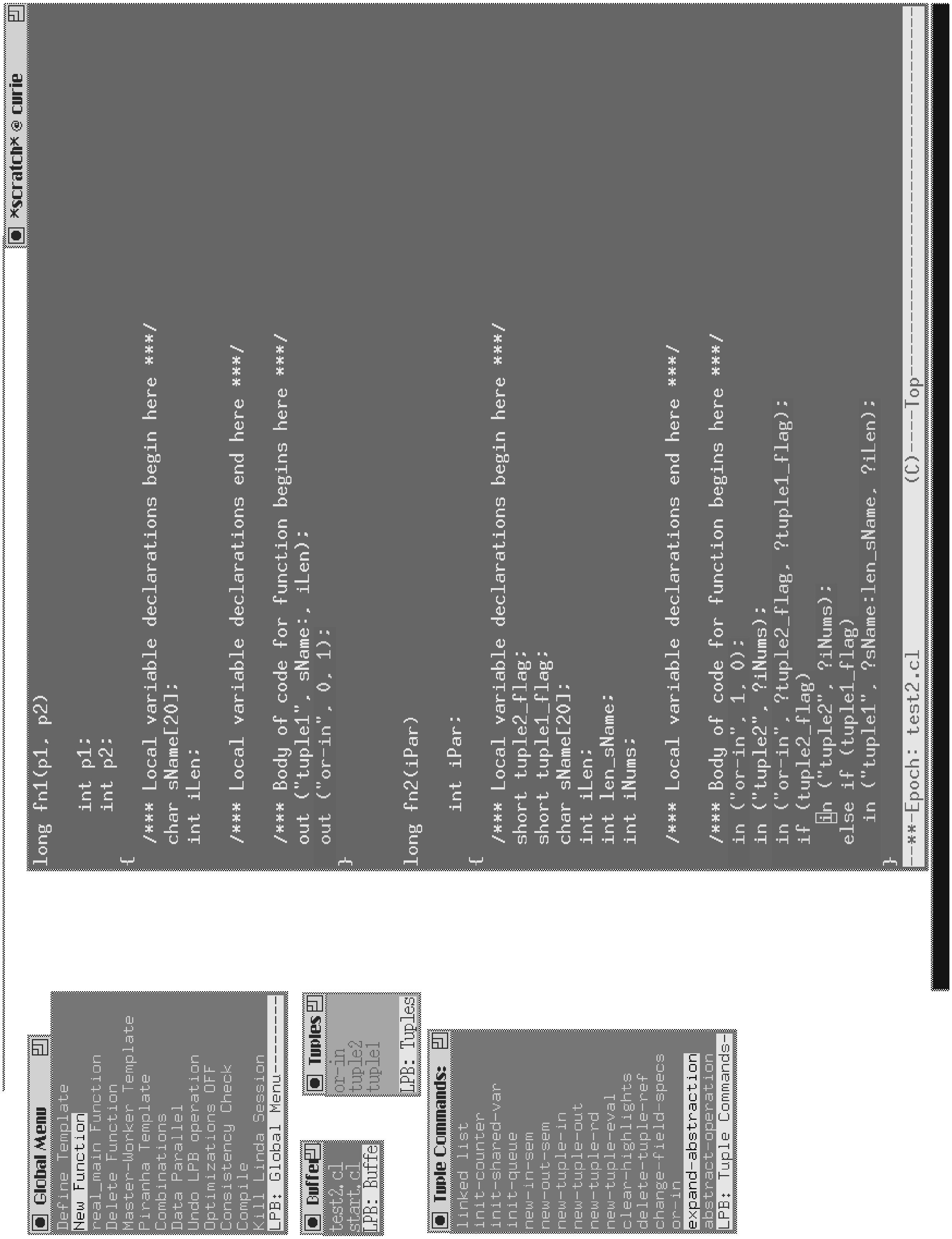

Figure 4: Expanded **or-in**

3 Interfacing to other tools

3.1 The compiler

The Linda pre-compiler parses and analyzes operations on tuples. Given certain sequences of operations, the pre-compiler may draw conclusions about the intended effects, but (like any compiler) it cannot in general infer the user's intent in specifying particular sequences of operations. The LPB has superior knowledge in this regard. Since the program is being constructed through templates or other higher level conceptual frameworks, the LPB "knows" why the various operations are being used. For example, the LPB knows that certain pairs of tuple operations are associated with "counter" tuples which will always be updated in certain ways. Given a particular distributed data structure, the LPB knows which tuple operations need to be used to create and manipulate it. The LPB has a deeper understanding of data structures than an analyzer can develop at compile time.

This knowledge is valuable to a compiler. Given an understanding of what a series of Linda operations is intended to achieve, the compiler can transform the series into a semantically equivalent series of operations that are more efficient. In some cases it will be possible to fuse these operations together and perform a smaller number of discrete operations over tuple space, saving overhead. We demonstrate this with a series of three examples of increasing complexity.

The first deals with the use of shared counters between processes. These are represented by tuples in tuple space with a field that contains the shared counter. Traditionally, any update to the counter would be accomplished by removing the tuple from tuple space, and introducing an updated tuple with the appropriate value incremented or decremented by one. In the context of the LPB, the user simply defines a counter using the relevant menu option, and thereafter selects an **increment** or **decrement** operation on this counter. The LPB then inserts the appropriate **in** and **out** commands into the code. Hence, a conventional compiler would understand this code in the same manner as if a user had typed in these commands from scratch.

Once the user has defined a counter, the LPB knows that apart from initialization, a counter update will always involve a pair of **in** and **out** operations. Every counter update is thus noted as a "special" tuple operation in a file. The compiler reads this LPB-generated information file and goes down the list of "special" tuple operations. The compiler places these tuples into their own partitions and marks them with special handlers, tailored to meet their specific needs. In particular, for the case of counters, the LPB actually marks the **in** operation in an increment or decrement as an increment or decrement, whereas the second operation is marked as a "no op." Hence, the first tuple operation in an increment or decrement gets assigned a particular handler, and the second is completely ignored.

The increment or decrement handler locks the counter variable in tuple space, updates it *in place*, and then releases the lock. This saves the compiler two locks and avoids an additional tuple operation. Knowing enough to throw away the second operation entirely is very useful. The original compiler (v2.4d [2]) was built to deal with only one tuple operation data structure at a time. For this optimization, we are actually dealing with two tuple operations at one time and hence, throwing out the second operation is particularly fruitful because we use the data from the first operation only. If we were to need data from the first operation too, we would need to modify the compiler to handle data from two or more tuple operations at one time, a change that proved to be necessary for other optimizations.

This optimization of counters show promising results. For a sequence of ten thousand counter increments and decrements on a Sequent Symmetry with 16 processors, the v2.4d compiler took 7.55 seconds. The new LPB-optimized compiler took 3.07 seconds for the same sequence. The time for a counter update was thus reduced by more than a factor of two. Since we not only eliminate an entire operation in this optimization, but reduce space and overhead constraints for even the single operation, this improvement by better than a factor of two is no surprise. Running the same test on a Sun Sparcstation 2 and Sparcstation 10 yielded almost identical factors of improvement.

A similar optimization was implemented for general shared variables, although this case is slightly more complex. Since the variable type is not a mere counter, but could be any structure or even function call, the second operation in the **in-out** pair cannot be ignored. The implementation of this optimization faced some problems that arose because the v2.4d compiler was not implemented with an LPB interface in mind. The compiler was designed to handle one data structure per tuple operation. For optimizations of the shared variable nature, we need access to two tuple operation data structures simultaneously. Hence, implementing this optimization involved some tricky coding which would be eliminated when a new compiler is designed from scratch with an LPB interface in mind. At any rate, for a sequence of ten thousand shared variable updates on a Sparcstation 10, the

[2]The version number comes from the Linda compiler of Scientific Computing Associates, New Haven

v2.4d compiler needed 16.64 seconds, whereas the LPB-optimized one required 8.62 seconds. This improvement is again in line with expectations. A total elimination of the second operation would result in a reduction in time by at least a factor of 2. Since, however, the second operation could not be completely eliminated, our reduction is slighly less than a factor of two.

The optimizations for shared linked lists required some tricky modification of the compiler. A shared linked list basically has two levels. The first level is the index list, where each entry is represented by a tuple containing the index value and the next index value. The index value is the key into another tuple which actually stores the data element for that index. At the user level, the user simply selects options of the variety **initialize list**, **insert node** or **remove node**. The user actually views the code at this level of abstraction too. If the user chooses to expand the abstractions, they expand into Linda code that manipulates the linked list. Hence, an **insert node** will expand into code that first acquires a unique identifier for the node and then creates a new node. To do this, we **in** the node after which the new one will be inserted, modify the appropriate fields to insert a new node, and **out** the new node index. Then we can **out** the node with the data. The unoptimized (v2.4d) version will compile this expanded code, whereas the LPB-optimized compiler will compile a second file generated by the LPB.

The implementation details are beyond the scope of the paper. In manipulating a very large list with ten thousand insert and delete node operations on a Sparcstation 10, the v2.4d compiler required 50.51 seconds. The LPB-optimized compiler ran the same sequence in 27.13 seconds. This was, again, consistent with expectations.

3.2 Program visualization

The Tuplescope visualizer [4] is a graphical monitoring tool that presents a dynamic image of an executing Linda program. Tuples are represented on the screen and their movement to and from tuple space is displayed as the program executes. The LPB passes enough information to Tuplescope to allow a better organization of the display. For example, a counter tuple normally looks like every other tuple, but with LPB-generated information, Tuplescope can recognize counters: a counter has its own window with a displayed value that changes as the counter is updated.

Similarly for queues; instead of displaying bare tuples, a graphical representation of a queue is displayed, with the contents of the queue displayed as a list and head and tail of the queue clearly visible.

Figure 5 shows some of the enhanced visualizations involving counters, shared variables, queues, and linked lists. The information associated with each partition is color-coded according to partition type and variable status.

In the long run, the various templates the LPB offers could each have unique representations in Tuplescope. Piranha programs[5], for example, all follow a particular pattern which is related to general master-worker programs. The LPB is aware of what constitutes a Piranha program and which processes are "feeder" and "piranha" processes. The current implementation of Tuplescope was not designed with the LPB in mind. What we need is an open interface which would allow the LPB to talk directly to the compiler and Tuplescope. If Tuplescope and the Linda compiler were redesigned with more open communications interfaces, that would open numerous doors for further interaction with the LPB. In particular, an interface language which allows the LPB to pass values and graphic object descriptions directly to Tuplescope would be extremely useful.

4 Extending a base language with a program builder

How do we capture programming experience and provide special-purpose conceptual and practical aids to a programmer in a framework that can adapt with changing methodologies? We could (1) rely on the flexibity of the base language, (2) build libraries of subroutines, (3) offer syntactic support, possibly with the addition of (4) semantic support.

Choices (1) and (2) are outside our range of interest. For our purposes it is important that special-purpose constructs look to the programmer as if they were language features: syntax must be clean, and the compiler must be capable in principle of profiting from the programmer's decision to use these particular constructs. Further, we intend for our language extensions to incorporate an interactive graphical environment.

Syntactic and semantic support can be provided in two different ways. We might build a new language; alternatively we might build some sort of preprocessor, which recognizes our new constructs and translates them into some base language. The key difference (ultimately one of degree and not kind) is that the preprocessor targets a higher-level virtual machine than the compiler. Other things being equal, a higher-level virtual machine language is more human-readable and more portable than a lower-level one.

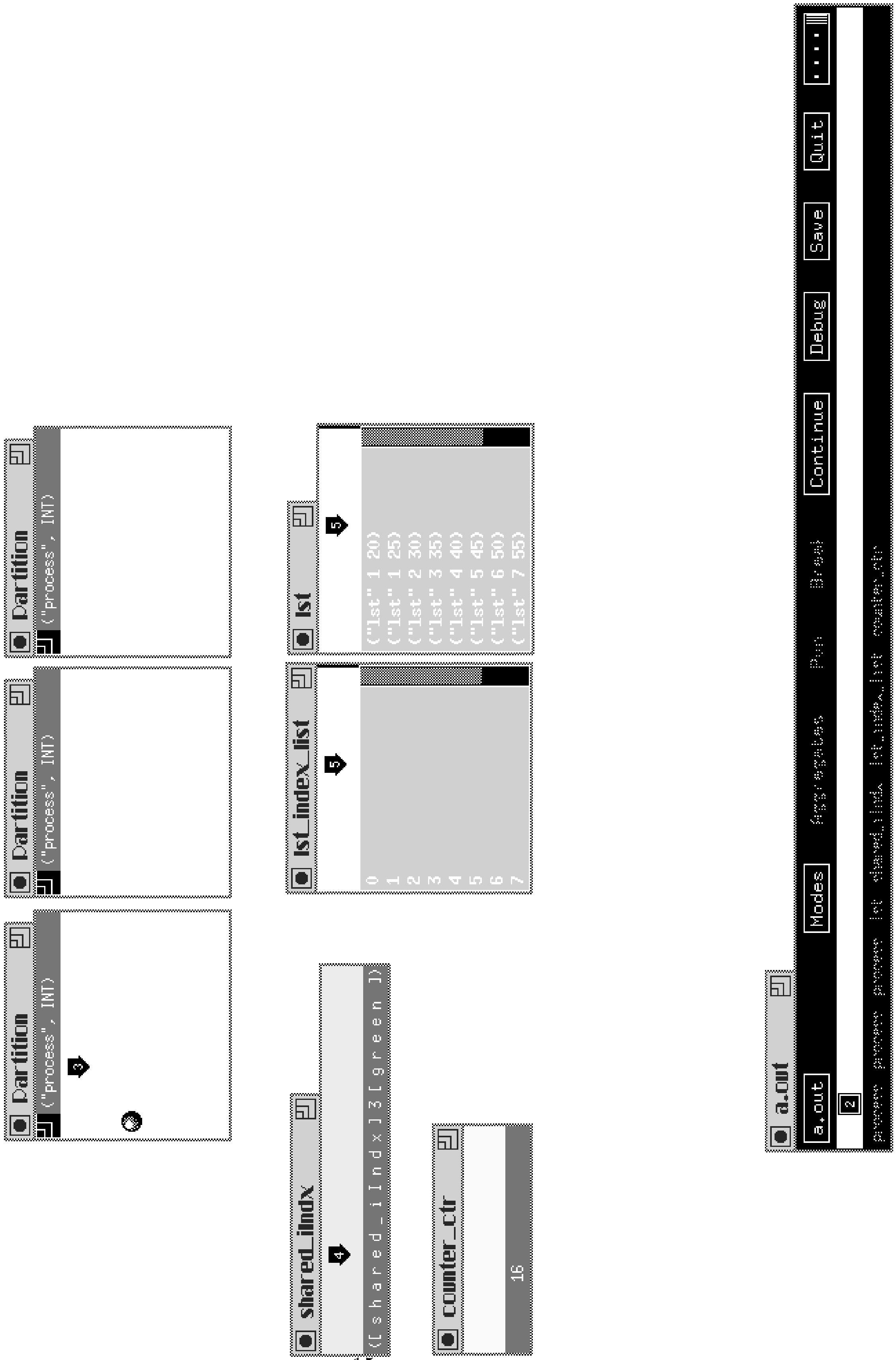

Figure 5: The Tuplescope enhancements

Concretely: the LPB is a preprocessor that targets C-Linda as a virtual machine; C-Linda is a family of compilers that ultimately produce object code for a particular machine, linked to a runtime communication library designed for a particular kind of interconnect and communication environment. C-Linda is a "higher-level virtual machine language" than the languages targeted by the C-Linda compilers themselves.

Hence when we have new language features to support, it is smarter in certain well-defined ways to implement them within the LPB than to design new languages. C-Linda is portable over virtually all asynchronous parallel environments, and the advantages of that are obvious. Those advantages, combined with the fact that people can read C-Linda more easily than lower-level target code, mean that the pre-processor environment is far more *customizable* and *dynamic* than a new language would be. Sites A and B (or even users A and B) can freely customize their preprocessors to support only and exactly the sort of constructs they need. The object code their customized systems emit is portable: hence they are excused, in their customization labors, from low-level hacking; just as important, no matter how much A's and B's preprocessors diverge, A can still run B's codes and vice versa. And no matter how much they diverge, A can still *read* B's codes, and vice versa. They share the same relatively human-readable target code. Focussing the same arguments on a single site, the preprocessor can be "dynamic": language features can be added, changed or deleted without requiring low-level re-implementations, and without destroying the compatibility or readability of older-version codes.

These advantages all depend, of course, on (1) the preprocessor's producing codes that are acceptably efficient, and (2) the preprocessor's being user-extensable in fact and not just in principle. Assuming these conditions are met, the preprocessor beats the new language option decisively for our particular needs: we suspect that "customizability" will be important in parallel and distributed programming environments, and we know for sure that "dynamism" is desirable.

The only remaining question is: what kind of preprocessor should we build? The LPB is an odd beast in preprocessor-land[3]...

The particular characteristics of the LPB style are: (1) *Input*: the LPB defines an environment—an interactive graphical one—and not merely a language. (2) *Output*: the LPB emits not only target code, but hints or directives to the compiler and visualizer. (3) *Trajectory*: the set of language features supported by the LPB is user-extensible.

[3] which is right next to Fantasyland in Disney World: check your maps.

When requiring input for a particular specialized operation, the LPB will interactively direct a user through a tree of options. In contrast, when using a conventional preprocessor, a user would have to make a routine call which is specific enough to represent the entire path down the tree. A conventional preprocessor can offer lots of such routine calls with large argument lists to create numerous powerful options, but it cannot provide *active* support during the construction phase such as guiding users through incremental program templates or menu-driven queue model selections with automated code insertion. There is an incremental methodology in the script of a program builder which a conventional preprocessor cannot duplicate.

At the output end of things, the interactive nature of the LPB and the continuous acquisition of semantic information enable it to pass useful information to other tools in the environment. In fact, the LPB could become a 'smart' assistant with the addition of an expert database system that is able to identify and call up either (a) templates that are likely to be useful for the type of program under construction, or (b) fragments of complete, real applications that are related in significant ways to the program under construction, and are thus potentially useful guides. We have developed an expert database system with exactly the right capabilities in principle [13]; retargeting it to the LPB is research for the near future.

The LPB language features are user-extensible in a number of ways. The template-building template is a mechanism to add new specialized operations and new frameworks for program construction. A user also has the option of implementing proposed language additions (e.g. the **or-in** function) which globally affect key code segments. The user can further specify how the new features will add to the LPB's semantic knowledge and provide directions on how much of this is passed on to the other tools. Hence, when adding a new customized operation, the user not only specifies how it transforms into the base *lingua franca*, but defines how it could be optimized and how it is to be visualized. This level of extensibility distinguishes the LPB from preprocessors (such as C++ in preprocessor guise [32]) that implement complete and self-contained languages.

5 Future work

The integration of the LPB with the compiler and Tuplescope is promising, but not as complete or developed as it could be. As we mentioned above, the v2.4d compiler and Tuplescope both preceeded the LPB by several years and were not constructed with the LPB in mind. What is needed is a clearly defined interface language that allows the LPB to communicate to the compiler which operations in the code are related, what variables are needed to optimize these operations, and how to carry out this optimization. The LPB would also communicate to Tuplescope how a particular construct is to be displayed, what variable values are needed for it, and what graphical constructs to employ in doing this.

The LPB's knowledge of program structure is characteristic of an expert-database approach to intelligent program development. The program database contains information that could be very useful in identifying program types and structures. Fertig has built an appropriate expert database system [13].

6 Related work

The LPB's most important template-based structure editor predecessor is the Cornell Program Synthesizer [28]. Unlike the synthesizer, however, the LPB does not enforce a rigid framework. Instead, the LPB captures methodologies and supports them, without imposing a strategy. What the LPB produces is source code, and the programmer is free to ignore or modify this as desired. This flexibility is essential to any expert programmer.

Other structure editor systems that are similar to the Cornell Program Synthesizer include Aloe [12] under Gandalf [17], MENTOR [16], CENTAUR [6] and IPSEN [22]. Like the LPB, KBEmacs [35] also used an Emacs-like environment with a higher-level programming environment on top. PSG [3] allows both structure and text editing, although the dynamic semantics of the language are defined in a functional language which is used to interpret the programs. Odin [10] investigates the idea that tools should be centered around a persistent centralized store of software objects, and Arcadia [34] is a large project investigating various issues involving an object management system. An interesting approach is taken by Enterprise [30], which expresses parallelism graphically, thus specifying communication and synchronization through a different medium.

Extensible parallel programming environments such as SIGMACS [31] generate a program database during compile time that can be used in later modifications to the program. The LPB, on the other hand, maintains a dynamic program-describing database that grows as the program is constructed. This allows the system to maintain semantic as well as syntactic information on the programs being developed. This information is used for guiding program development, for checking consistency, for documentation purposes, for providing optimizing information to the compiler [8], for enhancing graphical monitoring, and potentially also for benchmarking utilities to visualize performance in the spirit of [19].

There is currently much research effort in visualizing the dynamic behavior of parallel programs. [21] is a good example. DPOS [11] uses a metalanguage to define networks and provides a set of tools for visualizing and debugging. [24] is an interactive graphical environment for concurrent programming, using a specially-designed graphical language, Pigsty. Pigsty is based on CSP and Pascal, and currently limited to single processes and one-dimensional arrays of processes. Since the LPB can convey semantic information to a graphical monitoring tool [4], programmers can visualize dynamic information at a higher abstraction level than would otherwise be possible. There has been much work in the general area of visual systems such as PECAN [25], Garden [26], Use.It [18], PegaSys, [23], PICT/D [15], ThinkPad [29], or PT [20].

Some commercial environments such as HP's Softbench, ParcPlace C++, Lucid's Energize, or DEC's FUSE [14] use techniques similar to the ones in the systems mentioned above.

7 Conclusions

The LPB supports basic tuple operations as well as higher-level functions, and offers complete templates and program structures. It captures programming methodologies and guides the user through program development. As new methodologies emerge, they can be added to the LPB in the form of templates which are constructed via the the template-building template.

Further, the LPB's program-describing database can supply information to other tools in the environment, enabling optimization at compile-time, enhanced visualization at run-time, and eventually also performance monitoring for efficiency.

Most important, the LPB is characteristic of a potentially significant trend in programming language design. It addresses the traditional conflict between keeping a language simple and simultaneously demanding that it be higher-level. The proposed solution is to combine a simple, general coordination language with a higher-level, domain-specific system that provides the power and higher-level abstractions that a programmer can selectively choose to employ. In sum, we can have our cake and eat it too. If we can capture the methods and idioms that skilled programmers rely on *without* complicating the language itself with a galaxy of high-level, special-purpose constructs, we have a solution to an important problem.

References

[1] Shakil Ahmed, Nicholas Carriero, and David Gelernter. The Linda Program Builder. In *Proc. Third Workshop Languages and Compilers for Parallelism (Irvine, 1990) (invited paper).* Languages and Compilers for Parallel Computing II, MIT Press, 1991.

[2] Shakil Ahmed and David Gelernter. A CASE Environment for Parallel Programming. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering.* IEEE Computer Society Press, July 6-10 1992.

[3] R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, October 1986.

[4] Paul A. Bercovitz and Nicholas J. Carriero. TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs. Research Report 782, Yale University Department of Computer Science, April 1990.

[5] R. Bjornson, N. Carriero, D. Gelernter, D. Kaminsky, T. Mattson, and A. Sherman. Experience With Linda. Research Report 866, Yale University Department of Computer Science, 1991.

[6] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGPLAN Notices 24, 2*, February 1989.

[7] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs.* The MIT Press, 1990.

[8] Nicholas Carriero and David Gelernter. A Foundation for Advanced Compile-time Analysis of Linda Programs. Technical report, Yale University Department of Computer Science, 1991.

[9] Nicholas J. Carriero and David H. Gelernter. Linda in Context. *Communications of the ACM*, April 1989.

[10] G. Clemm and L. Osterweil. A Mechanism for Environment Integration. *ACM Transactions on Programming Languages and Systems*, January 1990.

[11] John D. Evans and Robert R. Kessler. A Metalanguage and Programming Environment for Parallel Processing. *LISP and Symbolic Computation: An International Journal*, 1991.

[12] P.H. Feiler and R. Medina-Mora. An Incremental Programming Environment. *IEEE Transactions on Software Engineering*, September 1981.

[13] Scott Fertig and David Gelernter. A Software Architecture for Acquiring Knowledge from Cases. In *Proc. of the International Joint Conference on Artificial Intelligence*, August 1991.

[14] Mary Jo Foley. Can We Talk. *SunExpert*, January 1992.

[15] E.P. Glinert and S.L. Tanimoto. Pict: An Interactive Graphical Programming Environment. *Computer*, November 1984.

[16] V. Donzeque gouge et al. Programming Environments based on Structured Editors: The Mentor Experience. In *Interactive Programming Environments, Barstow, Shrobe and Sandewall.* McGraw-Hill, 1984.

[17] A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, December 1986.

[18] M. Hamilton and S. Zeldin. Higher Order Software - Methodology for Defining Software. *IEEE Transactions on Software Engineering*, March 1976.

[19] Michael T. Heath and Jennifer A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, September 1991.

[20] Hsia and Amber. Programming through Pictorial Transformations. In *Proceedings of the IEEE International Conference on Computer Languages 1988.* IEEE Computer Society Press, October 1988.

[21] James A. Kohl and Thomas L. Casavant. Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems. In *Proceedings of the Fifth International Parallel Processing Symposium.* IEEE Computer Society Press, April 30 - May 2 1991.

[22] Claus Lewerentz. Extended Programming in the Large in a Software Development Environment. *ACM SIGPLAN Notices 24,2*, February 1989.

[23] M. Moriconi and D.F. Hare. The PegaSys System: Pictures as Formal Documentation of Large Programs. *ACM Transactions on Programming Languages and Systems*, October 1986.

[24] M.-C. Pong. I-Pigs: an Interactive Graphical Environment for Concurrent Programming. *The Computer Journal*, August 1991.

[25] S.P. Reiss. Pecan: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering*, March 1985.

[26] S.P. Reiss. Garden Tools: Support for Graphical Programming. In *Advanced Programming Environments, Lecture Notes in Computer Science Nr.244*. Springer Verlag, 1986.

[27] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator : a System for Constructing Language-based Editors*. Springer-Verlag, 1989.

[28] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, 1989.

[29] R.V. Rubin, E.J. Golin, and S.P. Reiss. ThinkPad: A Graphical System for Programming by Demonstration. *IEEE Software*, March 1985.

[30] Jonathon Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The Enterprise Model for Developing Distributed Applications. Research Report, Department of Computer Science, University of Alberta, 1993.

[31] Bruce Shei and Dennis Gannon. SIGMACS A Programmable Programming Environment. In *Proc. Third Workshop Languages and Compilers for Parallelism (Irvine, 1990)*. Languages and Compilers for Parallel Computing II, MIT Press, 1991.

[32] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.

[33] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, pages 419–490, October 1986.

[34] R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. *ACM SIGPLAN Notices 24, 2*, February 1989.

[35] R.C. Waters. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering*, November 1985.