THE SIMPLE SIMON PROGRAMMING ENVIRONMENT: A Status Report*

Janice E. Cuny *Duane A. Bailey*
John W. Hagerman *Alfred A. Hough*

COINS Technical Report 87-22
June 1987

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
USA

*The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research, contract N000014-84-K-0647.

Abstract

The Simple Simon Programming Environment is designed to serve as a testbed for the development of programming support specific to highly parallel computation. It consists of a set of programming tools together with a kernel that facilitates the prototyping of new tools. Currently, we have implemented prototypes of five tools: a textual language for database entry; a preprocessor for the convenient specification of nearly homogeneous code; a graph editor for describing process interconnection structures; a graph assistant for creating aesthetic graph layouts; and a "pattern-oriented" parallel debugger. We briefly describe each of these tools and their use in future research.

The Simple Simon Programming Environment is designed to serve as a testbed for the development of programming support specific to highly parallel computation. Such support might include mechanisms for the specification of multiple processes, the control of interprocess communication and synchronization, the mapping of logical structures onto physical interconnections, and the debugging of closely coupled processes for both correctness and performance.

Since the *process of programming* itself changes as more sophisticated tools and methodologies become available, high-level programming environments must evolve with use [1,2]. We have begun with a rudimentary – but extensible – parallel environment, called Simple Simon that runs as the frontend to the Simon Multiprocessor Simulator[3,4]. Simon is an event-driven simulator of concurrent, cooperating processes that are statically specified and individually programmed. It can model a variety of communication disciplines, executing a parallel application program as if each constituent process was running on its own processor. The Simple Simon environment makes Simon accessible to a local community of students and researchers. In turn, these programmers form the target community for the design and evaluation of successively higher-level extensions, making Simple Simon the testbed for our research.

Simple Simon consists of a set of programming tools together with a kernel that supports the prototyping of new tools. The kernel provides the core facilities of the environment; it is briefly described in Section 2. The tools provide optional features of the environment that can be replaced or modified at the user's discretion. Currently proto-

types of five tools have been implemented: a textual database language (TDL) that enables the user to enter information directly into our database; a generic code preprocessor (GCP) that allows the convenient specification of nearly homogeneous processes; a graphical database editor (GDE) that provides facilities for describing programs graphically; a graph assistant (GA) that uses heuristics to assist the programmer in designing aesthetic displays of interconnection structures; and a "pattern-oriented" debugger (Belvedere) that provides animations of system behavior. We describe program specification tools in Section 3, reserving Section 4 for discussion of the debugger. We summarize the contributions of Simple Simon in Section 5.

2. The Simple Simon Kernel

Our kernel augments the Simon Multiprocessor Simulator so that it can be used as the basis for our testbed. As shown in Figure 1, it has three components: Simon extensions that permit the use of hierarchies of shared memory; a graphics library that provides common interfaces at a variety of levels; and a database that contains all of the information known about an applications program. We discuss the database in more detail.

The Simple Simon Database

Existing programming environments for sequential or distributed computing are often integrated around a database that serves as the central repository of all information known to the environment. The Poker Parallel Programming Environment [5], for example, uses a relational database in the specification and representation of programs.

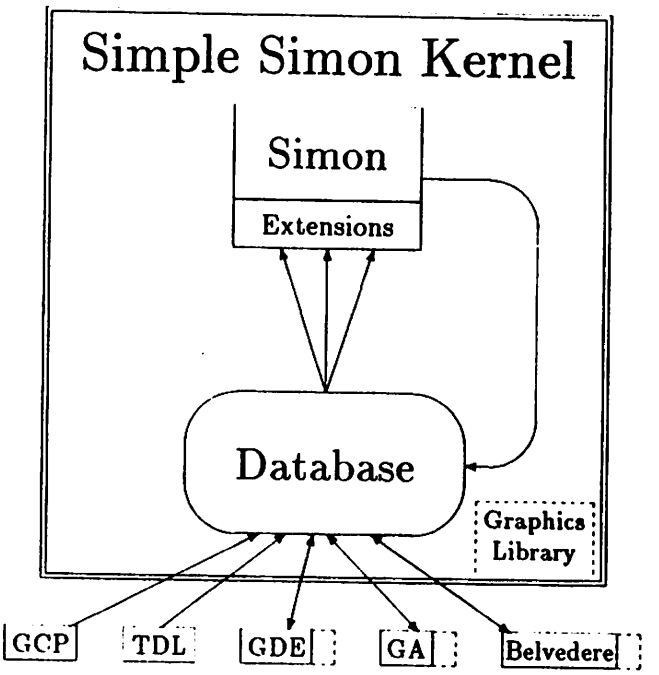


Figure 1: The Simple Simon Testbed. The kernel is shown within the double lines, while existing tools are shown outside the double lines. Arrows indicate the transfer of data. Dashed boxes associated with tools indicate use of the graphics library.

Our database contains both static and dynamic information. Static information describes program structures and consists of program segments, process interconnections, port and channel declarations, and a variety of spatial information defining the intended graphical display of objects. Dynamic information is added to the database as a simulation executes; it consists of a record of each simulated event, including primitive events (such as the get or put of a message) and high-level user-defined events (such as the exchange of values between adjacent rows or columns of an array).

Two views of the database are supported. The first is a high-level interface (used by most tools), providing object-oriented access to processes, channels, ports, messages and user-defined events. The second is a low-level interface providing direct access to the event stream produced by the simulator. Within the event stream, static information is recorded as events occurring at *Time 0* and dynamic information recorded as events occurring after *Time 0.*

3. Program Specification Tools

Simple Simon currently provides the user with four program specification tool prototypes. The textual database language, TDL, enables the user to enter information into the database in a traditional textual manner. It was designed as a primitive mechanism for use during the development of our first tools and, as a result, we expect that it will quickly become obsolete. The other three prototypes - GCP, GDE and GA - were designed to address more significant issues of parallel program specification.

Highly parallel programs in our environment differ from sequential programs in two important respects: they code segments for a large number of processes and they contain descriptions of the logical interconnections between those processes. Our preprocessor - GCP - addresses the first of these, i.e., the multiplicity of processes, by providing mechanisms for the replication and specialization of programs for nearly homogeneous processes. The graph editor and the graph assistant - GDE and GA - address the second of these, i.e., the description of process interconnections. GDE provides a graphics-oriented database editor for the specification of labeled graphs representing interconnection structures. GA provides facilities for creating aesthetic graph layouts.

Each of the four tools will be described. For the purposes of exposition, we consider the preprocessor first.

3.1 *The Specification of Nearly Homogeneous Parallel Programs*

Code for highly parallel programs often appears to require a large number of identical processes; on closer examination, however, many of these processes can be found to differ slightly because of initialization and termination details, timing, and edge effects caused by boundaries of the processor array. The result is that the programmer must potentially write a large number of similar programs, often duplicating his efforts. Previous attempts to avoid duplication have involved conditional execution of code segments [5,6]. Our approach is to provide a macro preprocessor for specializing code at compile time.[1] Process

[1] In the context of Simon, this preprocessor has the added advantage that it automatically generates the necessary, cumbersome channel names for interprocess communication.

specialization can be based on unique IDs, defined ports or user defined parameters. Figure 2 shows an example of preprocessor code. The segment is part of a gridsort program [7] designed for octagonal meshes (Figure 3); it shows the conditional generation of two comparison swaps, the first between pairs of adjacent rows of processes and the second between pairs of adjacent columns of processes. In the first swap, processes in even rows send their value `north` to be compared in the odd row process above; the smaller of the two values is then sent back. The variable `row` is a parameter giving the row in which a process resides. The code generated for all even row processes – except those on the top of the array – consists of just the `get/put` pair shown; processes on the top of the array do not have a `north` port defined and so (as a result of the first `ifdef` statement) do not participate in this swap. (This is an example of an "edge effect" mentioned earlier.) The code generated for all odd row processes – except those on the bottom of the array – consists of the `put/get` pair together with the compare and exchange statements shown; processes on the bottom of the array do not have a `south` port defined and so do not participate. In the second swap, the code segments are generated for even and odd columns of processes in a similar manner. This generic code segment produces sixteen different program segments, two of which are shown in Figure 4.

3.2 *Textual Specification of Process Interconnection Structure*

TDL provides a mechanism for entering descriptions of process and interconnection structures directly into the database. It allows the user to specify processes, ports, and

```
        .
        .
        .
#if (row%2 == 0)

#     ifdef north
          PUT("EVENROW",north,param(datum));
          GET("EVENROW",north,param(datum));
#     endif

#else

#     ifdef south
          GET("EVENROW",south,param(your_datum));
          temp=MIN(your_datum,datum);
          PUT("EVENROW",south,param(temp));
          datum=MAX(your_datum,datum);
#    endif
#endif
#if (col%2 == 0)

#     ifdef west
          PUT("EVENCOL",west,param(datum));
          GET("EVENCOL",west,param(datum));
#     endif

#else

#     ifdef east
          GET("EVENCOL",east,param(your_datum));
          temp=MIN(your_datum,datum);
          PUT("EVENCOL",east,param(temp));
          datum=MAX(your_datum,datum);
#    endif
#endif
        .
        .
        .
```

Figure 2: Preprocessor code segment prior to specialization.

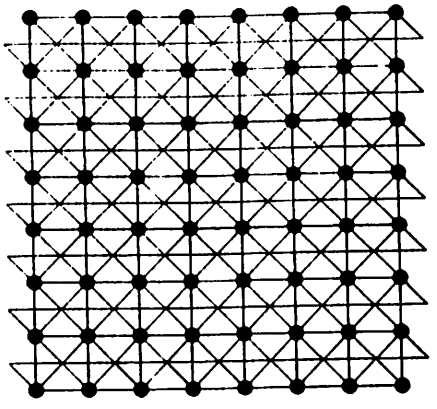


Figure 3: Octagonally connected 8 × 8 mesh of processors.

```
    ...
PUT("EVENROW",north,param(datum));
GET("EVENROW",north,param(datum));
GET("EVENCOL",east,param(your_datum));
temp=MIN(your_datum,datum);
PUT("EVENCOL",east,param(temp));
datum=MAX(your_datum,datum);
    ...
```

(*a*)

```
    ...
PUT("EVENCOL",west,param(datum));
GET("EVENCOL",west,param(datum));
    ...
```

(*b*)

Figure 4: Specialized code resulting from the generic code in Figure 2. (*a*) shows the code for a process that is an even row and an odd column in the interior of the array; (*b*) shows the code for a process that is in an even column in the first row of the array.

channels. In addition, it allows him to associate code segments with processes, processes with ports, and ports with channels. Figure 5 shows an example of a TDL program. It describes a mesh interconnection and uses preprocessor facilities for generating process IDs, process parameters and port definitions as described in the comments. Each port definition explicitly names the channel to which it connects and the type of access (`read`, `write` or `read/write`) it requires. While TDL is convenient for some structures, it is cumbersome for others, leading us to the development of graph editing tools.

3.3 *Graphical Database Editor*

Eventually, we expect that most input to our environment will be graphical but our initial version of GDE is limited to the description of process interconnection structures using labeled graphs in which the nodes represent processes and the edges represent logical communication channels.

Despite the fact that most existing parallel algorithms use one of a very few interconnection structures, we believe that the explicit description of interconnections is important. Such descriptions lend themselves to informative graphical displays, suggest information useful in the mapping from logical to physical structures [8], and provide redundancy for error analysis (making it possible, for example, to check for mismatched ports, dangling channels, path redundancy, or disconnected components). Explicit specifications enable the programmer to create the nonstandard graphs that are often needed when existing algorithms are composed into actual programs. In addition, they provide convenient mecha-

```
/* set up loop through processes */
#for row = 1,side,1
#for col = 1,side,1

/*  set up parameters related to processor location */
#   assign rowm1= row-1
#   assign rowp1= row+1
#   assign colm1= col-1
#   assign colp1= col+1

/* assign process ID, code segment and parameters */
   process gs'row'col {
     code "gs.c";
     param 'row, 'col, 'datum, 'side;

/* declare ports depending on where we are */
#    if (row>1)
         port north { channel ch'rowm1'col'row'col; access r,w;};
#    endif

#    if (row<side)
         port south { channel ch'row'col'rowp1'col; access r,w;};
#    endif

#    if (col<side)
         port east { channel ch'row'col'row'colp1; access r,w;};
#    endif

#    if (col>1)
         port west { channel ch'row'colm1'row'col; access r,w;};
#    endif

#endfor
#endfor
```

Figure 5: Segment of a TDL specification for a square mesh.

nisms for attaching labels (for example, processor id, code names, or parameters) to graph components.

Few existing systems permit explicit specification of interconnection structures; those that do rely on either textual input as in the definition of structured processes [6] or graphical input as in the Poker Parallel Programming Environment. We prefer the graphical approach. In Poker, the programmer is presented with a CHiP lattice [9] in which he connects processors by drawing the edges between them with cursor movements. The resulting display is appropriate for the CHiP architecture but not for the more general, logical structures that we describe.[2] In addition, the manual interconnection of processes is not feasible for massively parallel systems; because such systems are often designed in the small and then scaled, programmers need facilities for concisely describing *graph families.*

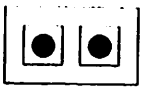
GDE is an early prototype, implemented so that we can experimentally determine the types of graph operations that are most useful in describing families of interconnection structures. Currently the editor has a textual input language that borrows features of Lisp but is not Lisp; the language is intended only for preliminary experimentation and we do not expect it to appear in future versions of the editor. Using GDE, a graph family is described by drawing its smallest instance and then modifying that instance into the next larger one. Commands used in the modification are recorded and then iterated to form larger family members. Thus, for example, the graph operations that transform the

[2]Currently, Simple Simon does not make a distinction between logical and physical structures. We expect to add such distinctions to Simon in order to investigate the properties of mappings from our logical specifications to actual hardware configurations.

three node complete binary tree of Figure 6 into the seven node tree of Figure 7 can be repeated to form any larger, complete binary tree. Figure 8 shows the full specification for the family of complete binary trees in which the appropriate iteration has been added by the programmer; this example shows the use of labels, labeling each node with its height (counting up from the leaves). We believe that many, if not all, useful graph families can be described in this iterative manner.

In our final system, we expect that input will be graphical and that the accompanying textual description will be produced automatically. For this reason, we have not been concerned with placement issues and we use a simple directional placement relative to *bounding boxes* (shown as rectangles in our figures). As nodes and subgraphs are added to the display, they are positioned using compass directions (`N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, and `NW`) relative to the largest bounding box on the screen. This scheme is not entirely satisfactory. Consider, for example, the alternative description of a complete binary tree description given in Figure 9. It builds the tree from the leaves upward by connecting adding a new root and then making a copy of the original tree to form the second subtree. For this description, our placement strategy results in the tree becoming increasingly skewed as it grows. We expect that these problems will be solved when we switch to graphical input.

Regardless of how the graph is described, its presentation can be changed with the help of our Graph Assistant.

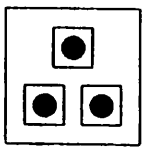


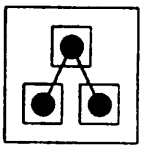
```
(setq tree (list (add-vertex N)))
```

(*a*)

```
...
(setq left tree)
(setq right (copy tree E))
```

(*b*)





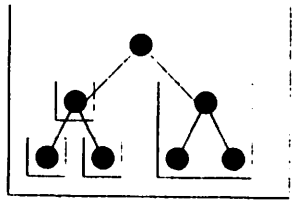
```
...
(setq root (list (add-vertex N)))
```

(*c*)

```
...
(connect left root)
(connect right root)
(setq tree (append root left right))
```

(*d*)

Figure 6: GDE description of a three node complete binary tree. The description starts with a single node created with the `add-vertex` function (*a*); this node is copied once to create a pair of siblings using the `copy` function (*b*); a root node is added using the `add-vertex` function (*c*); and the root is connected to its children using the `connect` function (*d*). The rectangles shown are *bounding boxes* used for spatial reference.

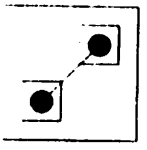


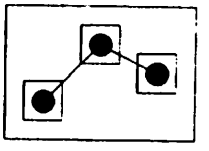
```
...
(setq left tree)
(setq right (copy tree E))
(setq root (list (add-vertex N)))
(connect left root)
(connect right root)
(setq tree (append root left right))
```

Figure 7: GDE description of a seven node complete binary tree constructed from the three node tree described above. Note that these instructions could be iterated to form successive members of the family of complete binary trees.

```
(setq tree (list (add-vertex N '((height 0)))))
(dotimes (i (- level 1))
   (setq left tree)
   (setq right (copy tree E))
   (setq height+1 (+ (extract (car tree) 'height)1))
   (setq root (list (add-vertex N '((height height+1)))))
   (connect left root)
   (connect right root)
   (setq tree (append root left right)) )
```

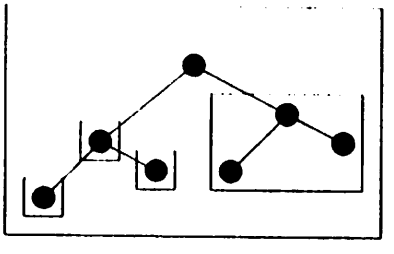
Figure 8: GDE description of the family of complete binary trees, parameterized by `level`, with nodes labeled by their height (from the leaves).





```
(setq tree (list (add-vertex N)))
(setq root (list (add-vertex NE)))
(connect root tree)
```

```
...
(setq right (pinned-copy tree E))
(setq tree (append root tree right))
```



```
...
(setq root (list (add-vertex NE)))
(connect root tree)
(setq right (pinned-copy tree E))
(setq tree (append root tree right))
```

Figure 9: Tree built from the leaves upwards. In the first step, a root has been added to a single node using the add-vertex and connect operations; in the second step, a right subtree has been added using a pinned copy of the original (now left) subtree. In the third step, these steps are repeated to form the next family member.

3.4 The Graph Assistant

The graph assistant supports the creation of aesthetic graph layouts. We anticipate its use in presenting structures that were described textually, in providing alternate displays of structures that were described graphically, and in displaying structures that were created dynamically.[3]

The Graph Assistant uses heuristics for creating appealing display that are invoked at the user's direction to place a graph on the screen. Figure 10 shows an example. The programmer inputs his interconnection structure - a mesh - to the database and then invokes the Graph Assistant. It's first attempt at a display is shown in Figure 10*a*. Interacting with the Graph Assistant, the user fixes the corners of the graph as shown in Figure 10*b* and then invokes a "pull" heuristic which minimizes the tension on the edges, creating the graph in Figure 10*c*. The final display, Figure 10*d*, is created with the use of a "equalize" heuristic which creates edges with uniform lengths.[4]

As another example, consider the cube connected cycle shown in Figure 11. It is shown as originally displayed in Figure 11*a* and after a single application of the "pull" heuristic in Figure 11*b*. At this point, the programmer manually "fixed" the nodes on the perimeter of the graph (as indicated by the filled in circles) and then iterated the "pull" heuristic to get the graph shown in Figure 11*c*. The final graph, Figure 11*d*, was achieved by manually

[3] At this time, Simon does not allow dynamic creation or deletion of processes but we intend to add the necessary extensions.

[4] The fact that the edge lengths were not entirely equalized in this simple example is an artifact of our implementation which we are in the process of remedying.

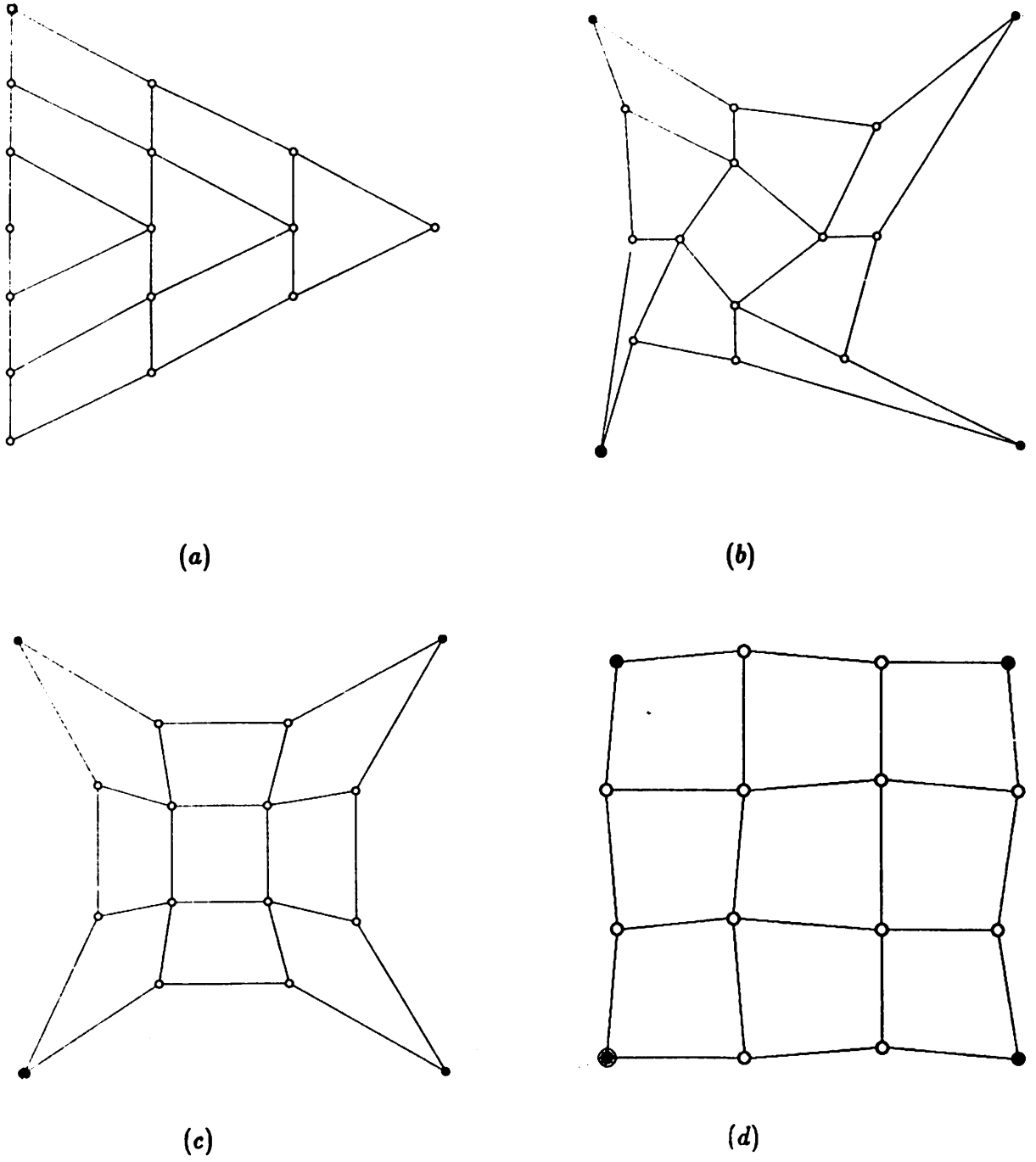


Figure 10: Placement of the nodes of a mesh using the Graph **Assistant.**

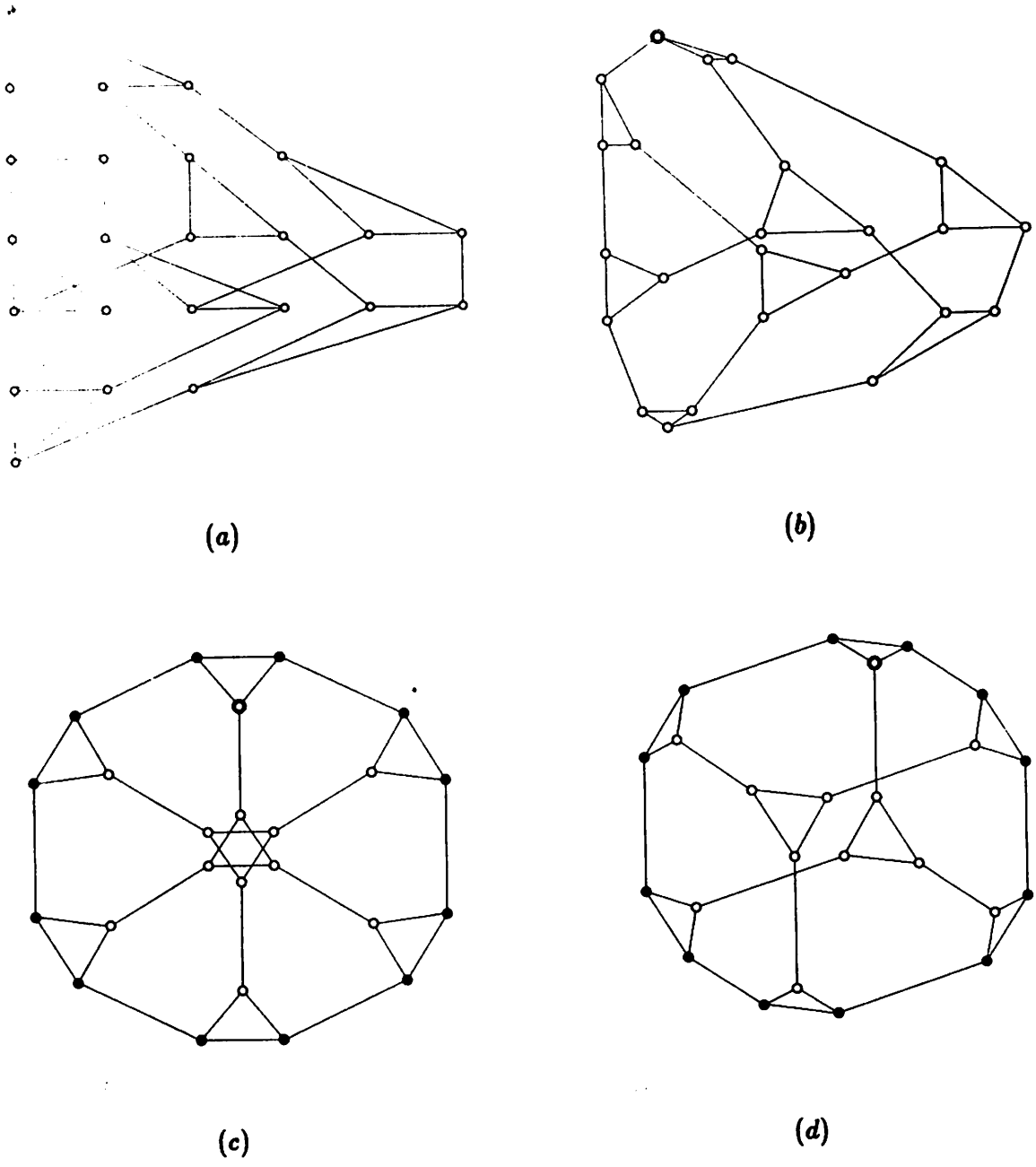


Figure 11: Placement of the nodes of a cube connected cycle using the Graph Assistant.

separating the inner triangles (and repositioning the fixed nodes slightly to get parallel cube edges).

We are currently investigating new layout heuristics, paying to particular attention to methods for exploiting symmetry.

4. Parallel Debugging

Highly parallel computation is not amenable to existing debugging techniques since parallel programs do not have the consistent global states, manageable quantities of potentially relevant information or reproducibility that have formed the basis for sequential debugging paradigms. Instead, their behavior is best understood in terms of the flow of data and control resulting from interprocess communication. These behaviors are often very structured: fine grain, tightly coupled processes communicate across regular interconnection networks resulting, at least logically, in patterned data and control flows. Since many of the errors in parallel programs are communication related, we believe that the identification of these patterns will form the basis for highly parallel debugging paradigms.

We have designed and implemented a "pattern-oriented" debugger, called *Belvedere.*[5] which we are using to investigate the animation and manipulation of interprocess communication patterns. Belvedere is a trace-based, *post-mortem* debugger [10,11] providing animations of program behavior. It operates on the event stream view of the database and provides animations of both primitive simulator events and high level, user-defined

[5] *Belvedere* comes from the Latin *bellus* meaning "beautiful" and *vedere* meaning "view."

events.[6]

Belvedere treats the event stream as a relational database. Users can select portions of the database for animation using standard database queries such as

```
add messages with event=READ-MESSAGE and time<61400
```

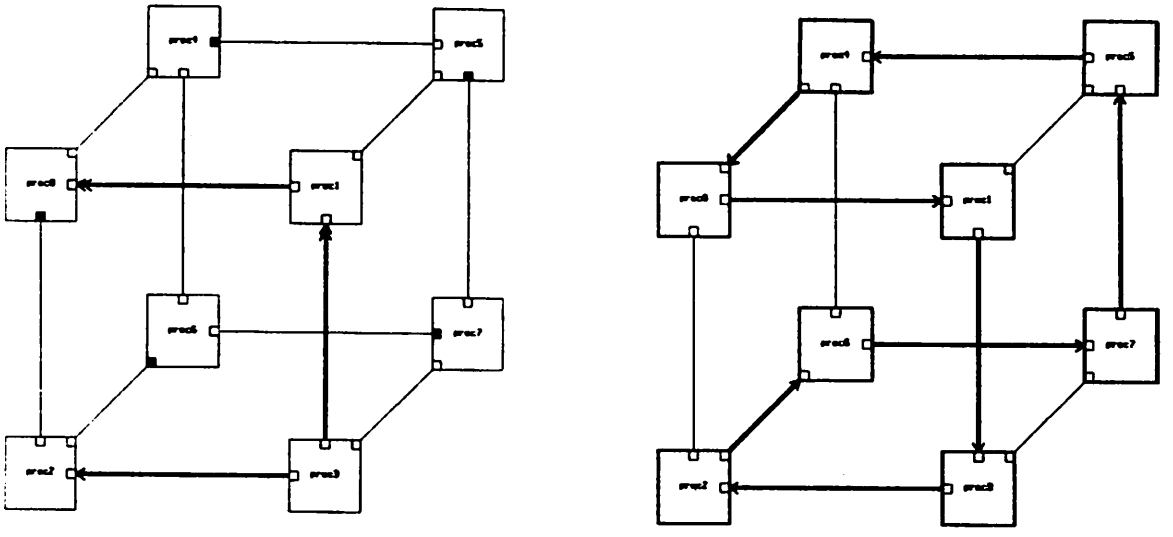
and

```
add messages with
        event=SEND-MESSAGE and message_name=conflicts
```

Each query results in the selection of a set of records which can be further modified by subsequent queries. At any time, the selected records can be animated. Figure 12 shows snapshots of sample animations of the above two queries. The animations were taken from the trace of a hypercube program implementing simulated annealing of the traveling salesman problem [12]. Each iteration of the program has two phases; in the first, pairs of processes communicate along cube edges in order to evaluate possible perturbations and, in the second, a ring synchronization is performed to implement accepted changes. Animated events are shown with highlighting. Figure 12(*a*) shows the first phase of an iteration using normal animation in which highlighting lasts only as long as the event itself; Figure 12(*b*) shows the path of a token during the ring synchronization using *traced* animation in which highlighting persists[7]

[6] While most existing systems do not provide such detailed information, its presence will allow us to determine which types of debugging information are most useful, perhaps providing insight for future hardware designs.

[7] To obtain these animations, with all of the processes and channels displayed, a set of background records were selected prior to each of the queries.



(*a*) (*b*)

Figure 12: Animations of a Simulated Annealing of the Traveling Salesman Problem. (*a*) Animation of processes communicating across cube edges during the first phase; (*b*) traced animation of the ring synchronization.

During animation, Belvedere automatically invokes event animators associated with each type of simulator event selected. The animators perform *closure* operations on the selected events in order to insure a coherent display. To depict a PUT-MESSAGE event, for example, the effected processes and channels must be displayed; thus, a closure operation is performed, selecting CREATE-PROCESS and CREATE-CHANNEL events for the needed objects; once these events are selected, their associated animators are automatically invoked.

While these simple animations are helpful, complex programs present patterns that are often difficult to interpret. Consider, for example, the snapshot of message traffic shown in Figure 13. It is taken from a *gridsort* program[7] in which an array of values (stored

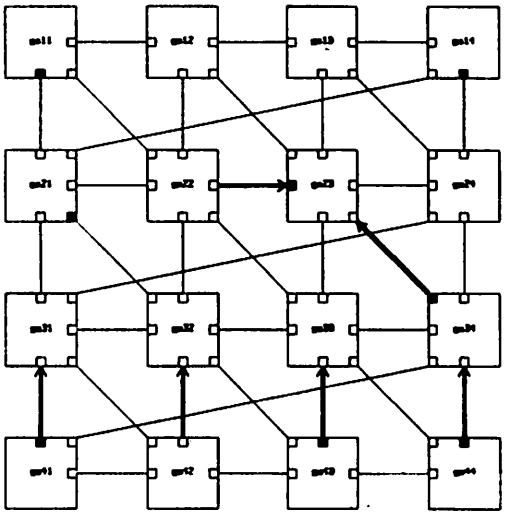


Figure 13: Snapshot of message traffic during a gridsort.

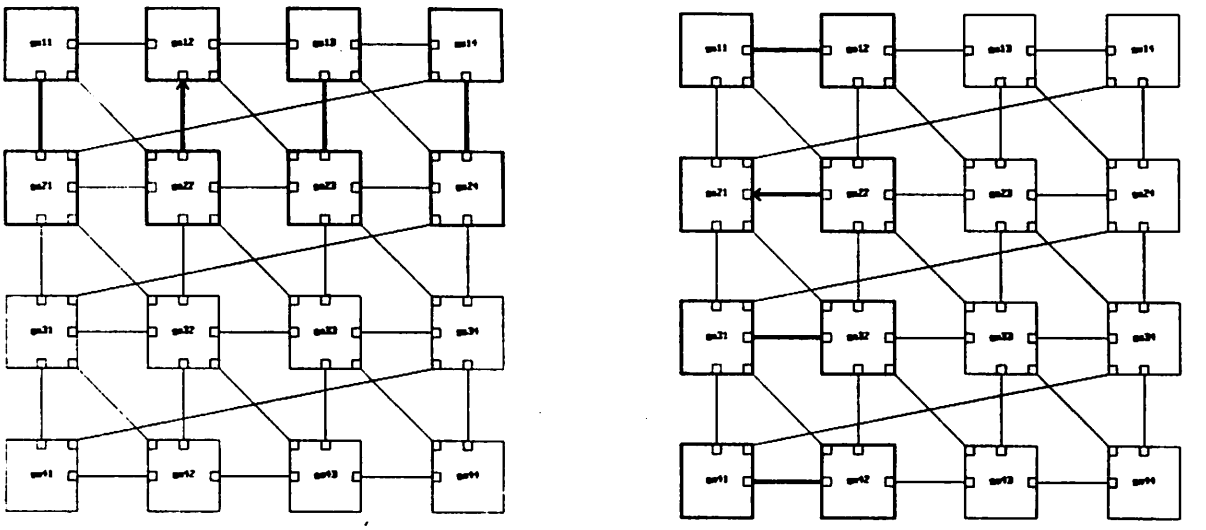
one per process) is sorted. During a single iteration, every process in the array does a

comparison-exchange with each of its adjacent neighbors. Logically, these comparisons proceed in lock step: even rows, even columns, even diagonals, odd rows, odd columns, odd diagonals. In the snapshot, however, the pattern is obscured by low-level details and asynchronous execution. Since the programmer understands his code in terms of sequences of row, column, diagonal swaps, it is important that he be able to determine the extent to which these patterns exist in the actual system behavior. To present that behavior in a way that matches his conceptual models, we have adopted the notion of abstract, user-defined events [13,14] and incorporated facilities to impose user-defined perspectives on those events.

The programmer can define abstract events in terms of patterns of lower level events. Once a high-level event is defined, instances of that event are identified within the event stream. Each instance constitutes a new event that is time-stamped and inserted into the event stream. Abstract events thus become available for selection by user queries.

Animating abstracts events requires some care. The programmer thinks of these events as occurring in an orderly manner because he thinks of them from a particular point of view, usually that of the individual process. In order to animate them in a manner that is consistent with his expectations, we provide the ability to define specific points of reference or *perspectives.*[8] The programmer could, for example, ask to see the gridsort's behavior from the perspective of processor (2,2) as shown in Figure 14. Note that he would then see the abstract row and column exchanges sequentially as he expected because processor

[8]Points of reference are conceptually useful but are also necessary to display concurrent overlapping high-level events in a coherent manner.



(*a*) (*b*)

Figure 14: High-level row (*a*) and column (*b*) events from the perspective of processor (2,2)

(2,2) participates in those events sequentially. (The animation of high level events from perspectives occurs during the time period that the point-of-reference participated in the event; during this time all locations (processors or channels) involved in the event are highlighted but only the component events involving the point of reference are animated.) Belvedere supports perspectives from processors, channels and data items.

Viewing the behavior of a program from a single perspective may well be misleading: from the perspective of process $(1,1)$, for example, the user would never see any ODD-ROW-SWAP, ODD-COLUMN-SWAP or ODD-DIAGONAL-SWAP events. In addition, the use of perspectives often sequentializes concurrent behavior possibly obscuring the existence of timing errors. To gain accurate insights into his program's behavior, the user will have to view it from a variety of perspectives. Since this is so easily done with Belvedere, we expect that it will be a valuable tool in determining discrepancies between actual and intended behaviors.

Belvedere is an early prototype and there remains a lot to be done. For example, Belvedere interprets system behavior according to user supplied patterns; patterns that occurred are displayed but there is no mechanism for reporting patterns that did not occur. In addition, it is well known that the best aids to debugging are often those encoded during the initial programming phase but we do not provide for such programmer input. Belvedere's animations are appropriate for nonshared memory, message passing systems but it does not provide support for shared memory architectures. And finally, Belvedere assists only with parallel programming errors; it does not provide any aid in detecting

errors in the sequential sections of code. We expect to address these shortcomings in future work.

5. Conclusions

We have presented a rudimentary parallel programming environment that operates as the frontend of a multiprocessor simulator and serves as the testbed for much of our research. The environment itself consists of a number of programming tools and a kernel which supports the development of new tools and prototypes. Within this testbed, we have begun to experiment with tools for the specification of nearly homogeneous process code, the explicit description of interprocess communication structures, the aesthetic display of graphs, and the debugging of parallel programs.

Acknowledgments

We would like to thank Mary Larson and Neville Newman for their contributions to the design of Simple Simon. In addition, we would like to thank Mark Bailey and Craig Loomis for their programming support, Mary Larson for her work on the preprocessor and Neville Newman for his extensions to the simulator. Finally, we would like to acknowledge Lawrence Snyder for his suggestions which resulted in a number of improvements in this text.

References

[1] L. J. Osterweil, "Toolpack - An Experimental Software Development Research Project," *IEEE Transactions on Software Engineering* SE-9(6), pp.673-685 (1983).

[2] W.Teitelman and L.Masinter, "The Interlisp Programming Environment," *Computer*, pp.25-33 (1981).

[3] R. Fujimoto, "Simon's User's Manual," University of California at Berkeley (1984).

[4] D. E. Heller, "Multiprocessor Simulation Program Simon," Shell Development Corporation (1985).

[5] Lawrence Snyder, "Parallel Programming and the Poker Programming Environment," *Computer* 17(7), pp. 27-37 (1984).

[6] Hungwen Li, Ching-Chy Wang and Mark Lavin, "Structured Process: A New Language Attribute for Better Interaction of Parallel Algorithm and Architecture," Proc. 1985 Int'l Conf. on Parallel Processing, pp. 247-254 (1985).

[7] Chip Weems, "Image Processing on a Content Addressable Array Parallel Processor," University of Massachusetts, COINS Technical Report 84-14 (September 1984).

[8] Duane A. Bailey and Janice E. Cuny, "An Approach to Programming Process Interconnection Structures: Aggregate Rewriting Graph Grammars," to appear Proc. of Conf. on Parallel Architectures and Languages Europe (1987).

[9] Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer* 15(1), pp. 47-56, 1982.

[10] R.M. Balzer, "EXDAMS - Extendible Debugging and Monitoring System," *Proceedings AFIPS Joint Spring Computer Conference,* 1969, pp. 567-580.

[11] H. Garcia-Molina, F. Germano, W.H. Kohler, "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering* SE-10(2), pp. 210-219 (March 1984).

[12] Edward Felten, Scott Karlin, and Steve W. Otto, "The Traveling Salesman Problem on a Hypercubic, MIMD Computer," Proceedings of the 1986 International Conference on Parallel Processing, pp. 6-10 (August 1986).

[13] Peter C. Bates, "Debugging Programs in a Distributed System Environment," University of Massachusetts, COINS Technical Report 86-05 (January 1986).

[14] Peter C. Bates and Jack C. Wileden, "High-level debugging of distributed systems: the behavioral abstraction approach," *Journal of System Software* 3, pp. 255-244 (1983).