Using a Template-Based Parallel Programming Environment to Eliminate Errors

Paul Iglinski, Nicholas Kazouris, Steven MacDonald, Diego Novillo, Ian Parsons, Jonathan Schaeffer, Duane Szafron, David Woloschuk

Department of Computing Science, University of Alberta, Edmonton, Alberta, T6G 2H1, CANADA

March 27, 1996

{iglinski, kazn, stevem, diego, ian, jonathan, duane, davidw}@cs.ualberta.ca

ABSTRACT

In this paper we describe how a template-based approach to writing distributed/parallel applications can be used to eliminate parallel programming errors. We take a two phase approach. First, the programming model can be designed to prevent many common parallel errors from occurring. Second, we show how an integrated set of tools that support the common model provided by the templates can be used to quickly detect and fix errors that cannot be prevented. In effect, a high-level view of a parallel program can be used to improve the software engineering properties of a distributed program, and reduce the time required to produce a correct, functional program.

This is a preprint of a paper that will be presented at the HPCS '96 Conference, Ottawa, June 1996.

1-INTRODUCTION

Many of the papers on distributed debugging begin with the message, explicit or implicit, that while debugging sequential programs might be characterized as a *difficult art*, debugging parallel programs is a decidedly *painful chore*. The cause of this drudgery is a combination of the special problems encountered in distributed computing and the lack of tools and methodologies for coping with these inherent difficulties. The search for solutions to this predicament can proceed from two distinctly different directions. One approach is to develop debugging tools to support existing parallel programming models, addressing the typical set of errors encountered by users of the systems. This approach is analogous to building a better mousetrap to trap those troublesome mice after they have entered a house. The other approach is to develop a programming model and parallel programming system (PPS) which preclude the introduction of errors. This approach is analogous to building a better house so that fewer or no mice can even enter.

In this paper, we discuss the advantages that template-based PPSs offer for the prevention and removal of parallel programming errors. The use of templates significantly reduces the possibility of programming errors by strictly managing the parallelism. In effect, templates provide a parallel structure framework within which the user supplies the missing application-dependent code. A major criticism of this approach is that by using generic structures, users lose the flexibility to tune their programs to achieve high performance. Although performance is the most often used (and abused) metric in the parallel/distributed computing literature, it is inadequate for the assessment of tools. In effect, template-based tools offer rapid prototyping and improved software engineering in return for (possibly) reduced performance.

There are three major sources for parallel programming errors:

1) Semantic errors: These errors are caused by a programmer's misconceptions about the parallel computing model being used and how to apply it to the problem at hand. For example, errors often arise from misunderstanding the semantics of memory shared between processes and the semantics of the parallel programming language used.

2) Implementation errors: These errors occurs because of the added complexity of parallel programs such as process launch, communication and synchronization. Typical errors include incorrectly packing/unpacking a message or creating a deadlock scenario.
3) Performance errors: These errors are caused by a lack of intuition or experience concerning the costs of parallelism and concurrency. For example, this class of errors includes executing fine grained program segments in parallel or poor synchronization choices that restrict concurrency.

The first two types of errors usually result in programs that execute incorrectly. The third source often leads to programs that run correctly but have poor performance gains when compared to sequential solutions.

This paper uses the Enterprise PPS [Schaeffer *et al.*, 1993] to argue that template-based models can be used to significantly reduce the time to produce a correctly working parallel program. The Enterprise programming model is built on top of the C programming language, providing a familiar base for writing programs (reducing semantic errors). A pre-compiler inserts all the communication and synchronization code, and the run-time system takes care of process-processor mapping and process interconnections (reducing implementation errors). The Enterprise meta-programming model allows the user to experiment with different parallel structures, often without changing the code (reducing semantic and implementation errors, enhancing performance and increasing user productivity). The environment offers a tool suite (animation, replay, debugging and performance monitoring) that is consistent with the programming model (helping address correctness and performance issues).

By placing much of the onus on the pre-compiler and runtime system, users are relieved of burdensome programming details, allowing them to concentrate more on the program design. By providing a seamless integration of the tools with the template model, programmers can recognize their errors at a high level and fix them within the context of the model.

Although template-based models are currently in vogue and many new models have recently appeared (for example, HeNCE [Beguelin *et al.*, 1993], PUL [Clarke *et al.*, 1994] and P3L [Bacchi *et al.*, 1994]) most concentrate on the programming model without consideration for support tools. We argue that tools such as debuggers and performance monitors should be integral to the design of the parallel programming system, and not just add-ons. Individually, many of the issues discussed in this paper are not new. What is important is how the design of the model pervades the entire system. This creates a uniform environment that is easy to use, reduces programming errors, and simplifies the detection of errors.

In this paper we refer to pre-packaged parallel behaviors as templates for compatibility with the literature. Recently however, object-oriented research has led to the development of a more general class of commonly occurring behavior classes called *design patterns* [Gamma *et al.*, 1994]. Each template mentioned in this paper can also be viewed as a design pattern.

In section 2 we discuss how template-based models can prevent many of the commonly-occurring parallel/distributed programming errors. In section 3 we describe how an integrated environment based on a common template-based model can be used to detect and correct those errors that cannot be prevented. Section 4 states our conclusions.

2-PREVENTING ERRORS USING A TEMPLATE-BASED MODEL

Many types of commonly occurring parallel programming errors can be prevented by providing a model which allows the compiler to handle most of the implementation details.

2.1-REDUCING SEMANTIC ERRORS

Most template-based models build on top of an existing programming language. In effect, the programmer uses sequential code (C in our case) to fill in the blanks of the template. By using a familiar sequential programming language, many of the semantic issues of new parallel programming languages (such as Orca [Bal *et al.*, 1992]), or extensions to existing languages (PAMS [Beltrametti *et al.*, 1989] for example) are eliminated.

2.2-PREVENTING COMMUNICATION ERRORS

Message passing is currently the preferred method for process communication in a distributed environment (the alternative, distributed shared memory, is still in its infancy). The traditional programmer's view of message passing involves four steps: 1) pack (marshal) the data, 2) send the message, 3) receive the message, and 4) unpack (unmarshal) the data. In some systems, such as PVM [Geist and Sunderam, 1992], these steps must be explicitly programmed in the user's code. Alternative systems, such as Sun RPC [Sun, 1986], only require the user to provide packing and unpacking routines. However, both of these low-level approaches require the user to write additional lines of code that only increase the probability of introducing an error. In contrast, Linda [Carriero *et al.*, 1994] handles the data transparently, but expects the programmer to explicitly call routines to do the communication. HeNCE [Beguelin *et al.*, 1993] allows the user to express this information graphically, but parameter information must be specified both in the interface and the code (causing a redundancy problem). Concert/C [Goldberg, 1993] eliminates the packing/unpacking routines by having the user provide additional type information describing the data. A better approach is provided by ABC++ [Arjomandi *et al.*, 1995] and Mentat [Grimshaw *et al.*, 1993] (for example), where the compiler does all of the work. In all these cases, higher-level tools not only reduce the programming effort but can also prevent programming errors.

Template-based PPSs provide templates that allow compilers to insert code to handle all four message-passing tasks transparently to the user. The Enterprise approach to message passing is to make RPCs look like standard C function calls, but without the programming effort and synchronous semantics of RPC. An Enterprise meta-program template alone determines that these calls are to be executed in parallel. Parallel Enterprise code looks like ordinary sequential code. For example, let's assume the procedure `Callee()` is to be executed in parallel with the procedure `Caller()` that is shown in Figure 1.

```
Caller()
{
        char a;
        USER_TYPE b;
        double d[100];
        int dnumb;
        int result[100];
        ...
        result[5] = Callee( a, b, &d[10], IN(dnumb) );
        ...
}
```

Figure 1. Communication in Enterprise

By letting the compiler generate the code to do data packing and unpacking, programming errors related to message packing are entirely avoided. Any heterogeneity concerns, such as byte-ordering schemes of the communicating processors, are handled automatically. Off-by-one errors in passing arrays are also avoided.

Notice that an additional parameter specifies the number of elements to pass (dnumb in the example). This approach is analogous to Fortran code which has the advantage of allowing the user to pass a portion of an array to reduce communication costs. Much like *Concert/C,* this size parameter is usually passed as the argument of one of three macros that indicate the direction in which array data is to be copied: from the caller to the callee (IN), from the callee back to the caller (OUT) or in both directions (INOUT). These macros are provided strictly as an efficiency mechanism to limit unnecessary data in messages. INOUT(), the default, will achieve the same semantics as C, allowing the array values in `Caller`() to be used and modified by `Callee`().

There is one important restriction on data passing in Enterprise. Although a pointer to an array or structure can be used as a parameter (along with the number of bytes to be passed), imbedded pointers are not allowed. To solve the imbedded pointer problem, extensive user annotations are required indicating the size and "shape" of the data (as in Concert/C). This restriction modifies the sequential semantics of C and creates a possible scenario for introducing logic errors. However, in our view, the benefit of not dealing with complicated and error-prone annotations significantly outweighs the disadvantages.

2.3-PREVENTING SYNCHRONIZATION ERRORS

In most parallel programming models, the user must specify the synchronization. For example, barriers and blocking message receives are common primitives that are explicit in the user's code. Since template-based models use standard sequential code, neither language extensions nor library calls are used for synchronization. That is, automatic synchronization techniques are provided that prevent synchronization errors from occurring.

Enterprise uses *futures* [Halstead, 1975] to synchronize concurrency so that the user can write sequential code with no explicit synchronization. Futures are becoming increasingly popular (see, for example, ABC++ [Arjomandi *et al.*, 1995] and Mentat [Grimshaw *et al.*, 1993]).

In Figure 1, `Caller`() continues executing without waiting for `Callee`() to return. Rather, it maintains all the return values, including OUT() and INOUT() parameters, as linked structures called futures. `Caller`() continues executing until `result[5]` is referenced in the subsequent code, at which time `Caller`() blocks if `Callee`() has not yet returned. As soon as the return message from `Callee`() is received, `Caller`() uses the return value as `result[5]` and resumes execution. Futures provide an error-free synchronization mechanism while routines without return values, and hence no futures, are totally asynchronous.

The richness of the template-based model determines the extent to which synchronization can be specified by the user, without resorting to the error-prone activity of inserting synchronization primitives manually. In Enterprise, parallel procedure templates are called assets. A replicated asset is a parallel procedure that can be concurrently executed on multiple processes. If a replicated asset is declared as *unordered*, meaning that the order of return values is irrelevant, it has fewer synchronization points, so concurrency is increased.

A trivial but instructive program, `CubeSquare` (Figure 2), illustrates how this automatic synchronization can be controlled by the user. This parallel equivalent of a "Hello world" program performs the trivial task of summing all the cubes and squares of numbers from 0 to `SIZE`-1. Assume that the `Square` and `Cube` routines are replicated assets so that several different processes can execute their code on successive calls. In this example, a series of calls are made to the same parallel functions and the order of the results is not critical, since the results are simply summed. Therefore the user can declare the replicated assets as having the unordered attribute. This attribute, as well as the replication factor, is part of the meta-program (template specification) and is independent of the user's code.

Unordered semantics denotes that the first reference to a particular return value for an asset will receive the first corresponding parameter value returned, regardless of whether the variable name matches or not. When the summation is executed in `CubeSquare`, the program may have to block and wait for `a[0]` and `b[0]`. However, if `Square` and `Cube` are unordered, other return values already available, perhaps `a[2]` and `b[3]`, may be used in their places. It is important to realize that the unordered attribute violates the semantics of sequential C, but can allow significantly more concurrency in certain computations less trivial than this example. Since this option, when used inappropriately, can lead to errors, it must be used with great care. Nevertheless, the user is guaranteed that synchronization errors will not result. The bottom line is that template-based models can provide the user with different levels of synchronization control while preventing synchronization errors.

```
#define SIZE      10

CubeSquare( argc, argv )
int argc;
char ** argv;
{
      int i, sq, cu, a[ SIZE ], b[ SIZE ];

      for( i = 0; i < SIZE; i++ )
      {
            a[ i ] = Square( i );
            b[ i ] = Cube( i );
      }

      sq = cu =0;
      for( i = 0; i < SIZE; i++ )
      {
            sq += a[ i ];
            printf("The square of %d is %d\n", i, a[ i ]);
            cu += b[ i ];
            printf("The cube of %d is %d\n", i, b[ i ]);
      }
      printf( "sum of squares %d\n", sq );
      printf( "sum of cubes   %d\n", cu );
}

int Square( i )
int i;
{
      SLEEP_RANDOM_TIME;/* Appropriately defined macro */
      return( i * i );
}

int Cube( i )
int i;
{
      SLEEP_RANDOM_TIME;/* Appropriately defined macro */
      return( i * i * i );
}
```

Figure 2. The code for CubeSquare.

2.4-PREVENTING PARALLEL STRUCTURE ERRORS

Most coarse-grained parallel programs conform to a small number of commonly occurring structures [Mehrotra and Pratt, 1982]. For example, pipelined, master-slave and recursive divide-and-conquer forms of parallelism are frequently seen. Some template-based tools provide these structures as building blocks for more complicated parallel programs (for example, PIE [Segall and Rudolph, 1985]).

With template-based PPSs, the parallelism is defined by using one or more templates from a fixed set. The code that creates the processes and establishes the parallel communication structure between them is created automatically by the PPS. Assuming that the PPS is implemented correctly, this guarantees that the parallel structure will be generated correctly. Furthermore, since there are no explicit references to the parallel structure in the code, the user cannot introduce structure errors when the code is later modified during debugging, performance tuning or maintenance.

For example, in Enterprise, the templates for the `CubeSquare` program of Figure 2 are expressed graphically in the Design View of the graphical user interface as shown in Figure 3. The user is assured that any program graph that can be drawn in Enterprise is syntactically correct, although it might not be an appropriate choice for the problem.

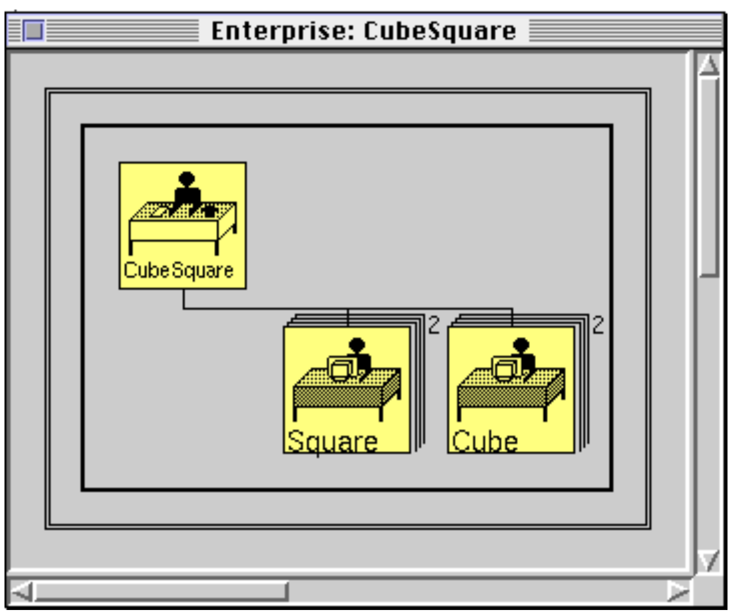

Figure 3. A meta-program for CubeSquare.

For the CubeSquare program, the templates consist of a department asset with three components: `CubeSquare`, `Square` and `Cube`. Each of these three assets is implemented by one or more processes that execute the appropriately named procedures of Figure 2. All of the code that spawns the processes and connects them together into this parallel "department" structure is automatically generated by Enterprise with no possibility of structure errors. In effect, the sequential code of Figure 2 and the asset diagram (parallel annotations) in Figure 3 are all that are needed to create a distributed Enterprise program.

Depending on the richness of templates available in the PPS, and the way they generate code, it may also be possible to change templates without modifying the user code in any way. For example, in Enterprise, it is possible to change the number of processes (called the replication factor) that can independently execute a parallel procedure without changing the user code in any way. In Figure 3, the superscripts on the `Square` and `Cube` assets indicate that two copies of each asset should be created. The programmer can change either or both of these values and, without changing the user code, use the new template to re-execute the program. In a non-template-based PPS, such changes would need to be made directly in the user's source code and errors could arise. In a template-based PPS such changes can be done without the possibility of introducing any errors.

Changing a replication factor is a fairly straightforward change to the parallel structure of a program. However, it should also be possible to make more dramatic changes in the parallel structure of a program and guarantee that no structure errors will be introduced. One of the important features of Enterprise is that one can experiment with parallelization techniques without necessarily having to modify the code. Enterprise supports combining assets in a hierarchical structure. For example, it is possible to have a department of pipelines (pipelines are called Line assets in Enterprise terminology). Each line asset could contain recursive divide-and-conquer assets (Divisions).

Having an orthogonal relationship between a program's code and its parallel specification dramatically reduces the possibilities for errors. Enterprise does not achieve complete orthogonality since changing between some parallel structures may involve moving code between various files. However, changes can be made in replication factors, ordered/unordered attributes, machine preferences (including different architectures), parameter and event logging specifications, and output windows without a single change to the program code. Once again, the result is fewer errors.

2.5-PREVENTING DEADLOCK

Deadlock is a common problem in message-passing programs. One of the conditions for deadlock is to have a cyclic dependency. In Enterprise, the communication graph is implicit in the user's template selection. Except for division assets (discussed later), the templates provided in the system preclude the creation of a cyclical communication graph among processes. This eliminates the possibility of a communication deadlock due to parallel structure. Of course, the user can still create deadlock by creating other cycles such as having a process waiting for an I/O event to occur.

It is possible to preclude deadlock without completely eliminating cyclic graphs. For example, in Enterprise, there is only one template that includes cyclic calls. It is a recursive template, called a Division, that is used to implement parallel recursion for divide-and-conquer algorithms. However, all recursive processes are created and controlled by Enterprise and communication deadlock cannot occur. Each time a procedure calls itself, the call either generates a new process or is a sequential call (in a parallel tree leaf node). Since a process need only wait for its children and not any of its siblings or parents, deadlock cannot occur.

2.6-PREVENTING HETEROGENEITY ERRORS

Enterprise includes its own internal *makefile* to automatically handle the issues of compiling and running processes on a collection of heterogeneous machines. The system maintains directories for each potential target architecture. This removes a troublesome bookkeeping burden from the user and eliminates another source of errors.

Enterprise also maintains a resource manager, which knows about the capabilities of each machine that can potentially be used in a computation. The system maps processes to processors based on any constraints specified by the user. The default is to put processes on idle machines. Users are not responsible for doing any process-processor mapping, unless they explicitly want to.

3-DETECTING ERRORS USING A TEMPLATE-BASED TOOL SET

A programming model, no matter how sophisticated, cannot prevent all types of errors. Although the Enterprise programming model can eliminate many of the errors that commonly occur when parallelism is introduced into an application, tools are still needed to identify and correct errors. Many parallel systems provide little support for debugging. The most common techniques of inserting print statements or attaching a sequential debugger to each concurrent process, are of little value in detecting errors that occur due to the interaction of two or more processes.

We divide the non-preventable errors into two classes, logic errors and performance errors. A logic error results in a program whose results are incorrect. A performance error results in a program that produces correct results, but executes too slowly. Different tools are required to detect and correct these different types of errors. In practice, logic debugging and performance debugging are not independent activities since performance tuning may introduce new logic errors into the code. Therefore, the PPS must provide a simple method of switching between logic debuggers and performance debuggers. The advantage of using a template-based PPS is that a high-level model exists. If both kinds of debuggers are based on the same model, then context switches between the two can be made more easily.

A uniform, fully integrated PPS such as Enterprise, allows the user to develop distributed programs with the advantage of built-in error reduction and debugging mechanisms. In Enterprise, a Design View provides the facilities for creating the program's parallel design (the meta-program), coding the assets, compiling the assets, and executing the program on selected workstations. An Animation View allows the user to animate the message-passing behavior of a program based upon a captured event file from a previous execution. In the Animation view, a program does not actually re-execute. However, logged messages can be examined and performance analysis tools can be invoked to graphically view performance statistics. A Replay View, similar in appearance to the Animation View, couples animation with facilities for deterministically re-executing a program under the control of a high-level event-based breakpointing tool. During replay, sequential debuggers can also be attached to selected processes. The Performance Views allow users to study the run-time performance of their programs. They simultaneously provide a macroscopic perspective of the program together with microscopic views of the state of individual processes.

3.1-SEQUENTIAL EXECUTION AND STANDARD DEBUGGING

Compiled programs can be run either sequentially or in parallel at the flip of a switch. Each asset (process) in Enterprise has the capability of being invoked by a message or by a sequential procedure call. Sequential mode simply disables the sending and receiving of messages. This allows the user to debug a program sequentially, before trying the program with parallelism enabled.

3.2-THE ANIMATION VIEW AND PERFORMANCE DEBUGGING

To take advantage of the facilities in the Animation and Replay Views, an event log must be recorded for a program execution. This log contains an abbreviated record of all the inter-process communications together with state changes, such as when a process blocks waiting for a reply or when a process terminates. The approximate total order of events captured in the log provides a basis for the performance analysis tools. A guaranteed partial order ensures a deterministic description of the program without tachyons (events out of their logical temporal sequence), such as a message being received before it is sent. This model forms the basis for execution replay and animation. The user has control over the amount of information logged at run-time and event logging can even be disabled for production runs.

Once an event file has been created, the Animation View can be used for both performance debugging and correctness debugging [Lobe *et al.*, 1993]. The captured events are animated as a sort of time-lapse movie. Messages are created by the calling assets, move to message queues and arrive at the called assets. In addition, the assets change state as execution proceeds. In the Animation View, the user can, in effect, simulate a particular execution, without actually re-running the program. The post-mortem analysis of a program in this view can reveal important performance characteristics and help to detect various programming errors. By viewing the animation, the user can dynamically observe the degree of parallelism, the relative states of processes, the buildup of messages in message queues, and the values of logged parameters in the messages. Figure 4 shows a snapshot of the animation for the `CubeSquare` program.

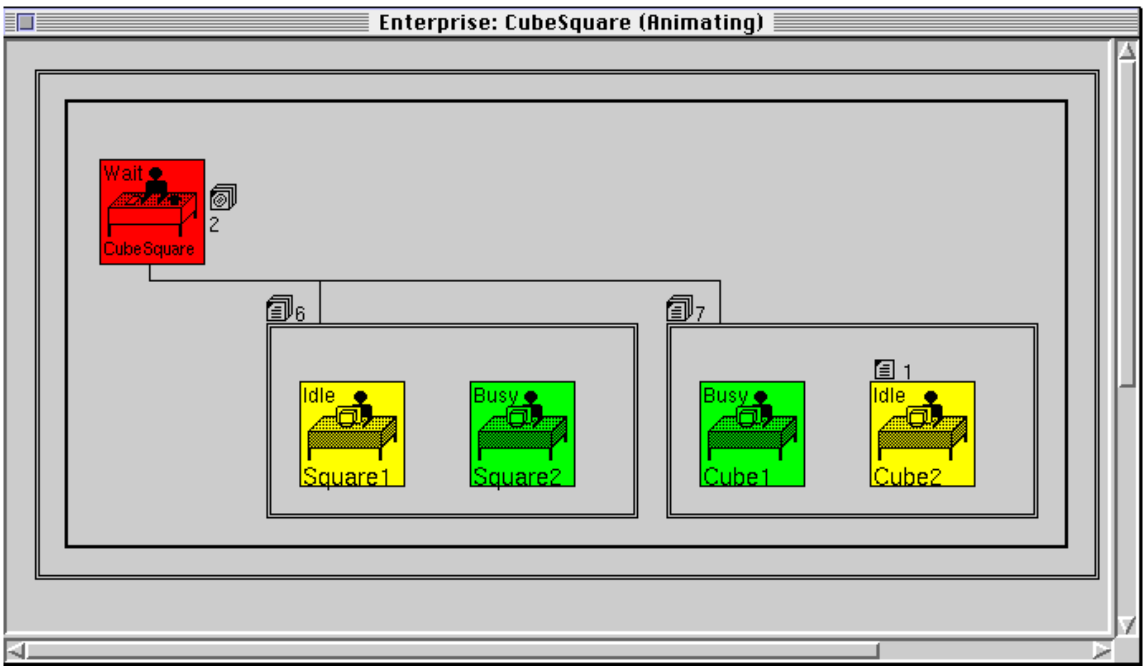

Figure 4. A snapshot of the animation for the CubeSquare program.

The Animation View can display the program's assets in fully expanded form, or with assets selectively collapsed to hide uninteresting detail. Each asset is labeled by its state: IDLE, BUSY, WAIT or DEAD. Each asset has two message queues: an input queue at the left of its top edge for call messages and a reply queue at the middle of its right edge for return messages. If the queues are empty, nothing is displayed. During animation, messages move along paths on the screen between assets and enter the message queues, which are then visibly displayed by a message icon and a number designating the number of messages in the queue. At any time, the animation can be stopped and the messages viewed. Since we have access to the compiler, each parameter in a message is displayed in a high level form including its name.

3.3-THE REPLAY VIEW AND CORRECTNESS DEBUGGING

Any tool that is used to detect and correct parallel logic errors must deal with two fundamental issues, non-determinism and the probe effect. In sequential programs, logic errors are almost always deterministic. In parallel programs, logic errors are often non-deterministic. The execution path may depend on race conditions between concurrently executing processes and the results of these races often depend on many factors such as processor load, network traffic and disk usage. If an error occurs on one of these paths, it may be necessary to execute the program many times (even hundreds or thousands of times) to exercise this error path once. In addition, it may be necessary to exercise the error path many times to isolate and fix the error. A good PPS must allow the programmer to reproduce any non-deterministic execution path as many times as necessary to isolate and fix the error.

The probe effect occurs when a programmer inserts debugging code into a program that alters the result of a race and changes the execution path so that an error is masked. When the debugging code is removed, the error re-appears. A good PPS must provide facilities to reduce the probe effect so that the debugging tools can isolate the errors.

Although logic errors in a program can sometimes be debugged by post-mortem examination in the Animation View, there are times when it becomes necessary to actually re-execute the program deterministically along the path captured in the event log. This is possible assuming that there are no non-deterministic constructs in the sequential code of the assets themselves. To facilitate a prescribed forced execution that allows the programmer to examine the internal state of individual processes and message contents not captured in the log, a Replay View was implemented for Enterprise [Iglinski, 1994]. This replay facility, a message-level breakpoint facility, and selective access to standard sequential debuggers all together constitute the Enterprise debugger.

Breakpoints are set at a high level, in terms of message-passing events. These breakpoints can be either unconditional for a particular event type at a particular asset grouping, or conditional upon the values of any parameters which have been captured in the event file. When a set breakpoint is triggered, the guided replay stops just *before* the event is executed. It is then a simple operation for the programmer to single step through the event, examine the contents of a logged message, or attach a sequential debugger to any process for lower-level debugging. Unlike other parallel debugging systems such as Node Prism [Sistare 94], the Enterprise debugger does not attach debugging processes to all nodes or to nodes with scheduled breakpoints, since the breakpoints are based upon captured information in the event file, not upon the internal states of processes. When a breakpoint is triggered and a suspicious process is identified, a sequential debugger can then be selectively invoked. Enterprise is scalable in that breakpoints can be associated with either complete sets of replicas or with particular nodes. Although the present implementation of Enterprise is not intended for the massive parallelism that Node Prism is designed to accommodate, the validity of the Enterprise model is not precluded in a massively parallel environment.

The specification and management of breakpoints is accomplished with a Breakpoint Browser (Figure 5) which is fully integrated into and coordinated with the display in the Replay View. Breakpoints are graphically depicted in the view by means of icons and highlighting. The text-based Breakpoint Browser is coordinated with the graphical Replay View while the breakpoints are being defined, and while the program is replaying and triggering breakpoints. Alternately certain structural components of breakpoints can be defined directly by selecting icons in the Replay View. These selections are reflected immediately in the Breakpoint Browser. This technique of coordinating textual information with graphical visualization is in accord with a core goal of Enterprise: to provide an intuitively comprehensible interface to an inherently complex and potentially confusing parallel architecture.

Although the Enterprise debugger lacks much of the power of a parallel debugging system like Node Prism, its virtue lies in its uniformity with the underlying model and the system into which it is integrated. It is easy to learn, simple to manage, and effective. Context switches between the Design, Animation and Replay Views are completely seamless. This uniformity increases the productivity of parallel programmers [Szafron and Schaeffer, 1996].

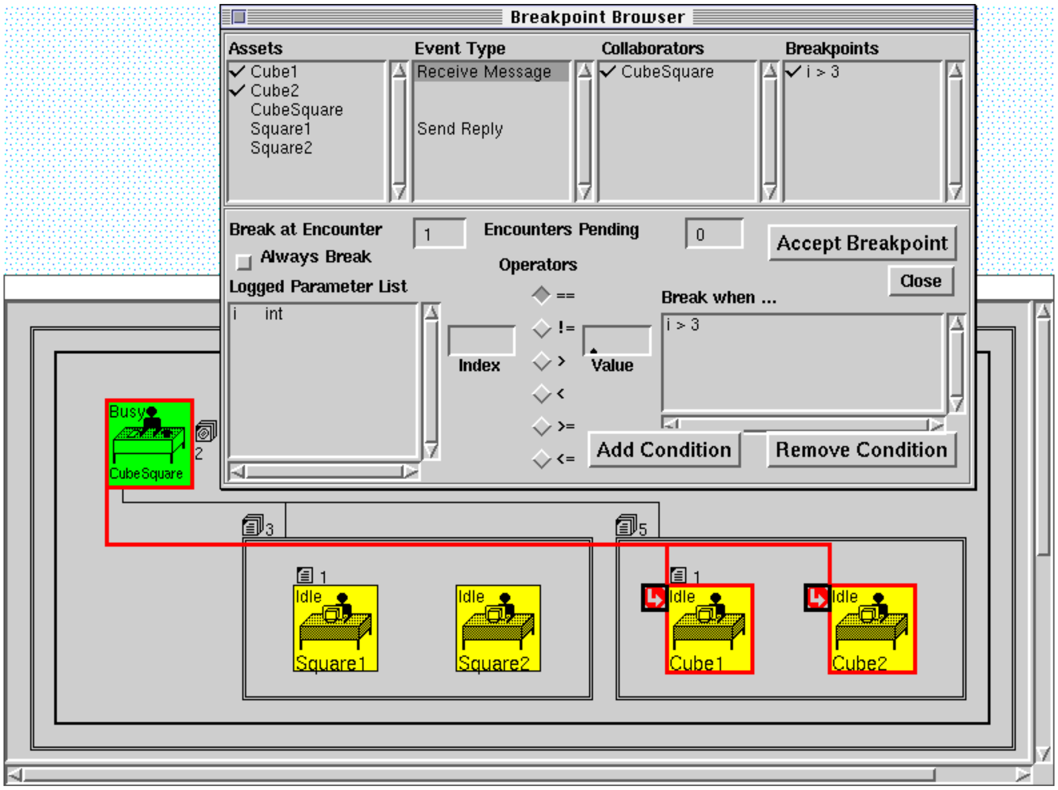

Figure 5. Breakpoint Browser and Replay View.

3.4-THE PERFORMANCE VIEWS

Performance debugging is quite different from logic debugging and there are different issues that must be addressed [Woloschuk *et al.*, 1995]. Performance tuning can be divided into three operations. The first operation is to *acquire* (capture or record) the interesting events while the program is running. The second operation is to *analyze* the events to produce information about the run. The third operation is to *present* this information to the user. There are two methods to acquire and analyze events. *Real-time analysis* is carried out at run-time as events are generated. In *post-mortem analysis*, events are recorded at run-time, but are analyzed in a post-execution process.

The two primary problems associated with real-time analysis are the number of events that can be generated by many concurrently executing processes and the difficulty of analyzing long-term patterns and trends in these events. The difficulty with post-mortem analysis is that an entire event trace must be captured and stored before analysis and presentation can begin, even if the interesting events happen at the start of a very long execution. However, both real time and post-mortem analysis are handicapped by fundamental limitations to the amount of information that can be displayed and by the amount of information that can be absorbed by human observers. Good PPSs must combine efficient event logging mechanisms, powerful analysis engines and good abstraction techniques for presenting information concisely.

The available Performance Views include the Asset Utilization view, showing how busy each asset is; the Transaction Time-line View, showing message transmission patterns; the Transaction Summary View, showing the details of each message in a transaction (sequence of messages); and the Annotation View, where the system attempts to comment on the parallel program's performance. Figure 6 shows a Transaction Summary View which can be used to trace a sequence of messages (a transaction) from asset to asset. It also allows a programmer to discover how much time each message spends in transit, waiting in queues and being executed. Finally, it describes the dynamic distribution of messages by reporting the minimum, maximum and average times of messages.

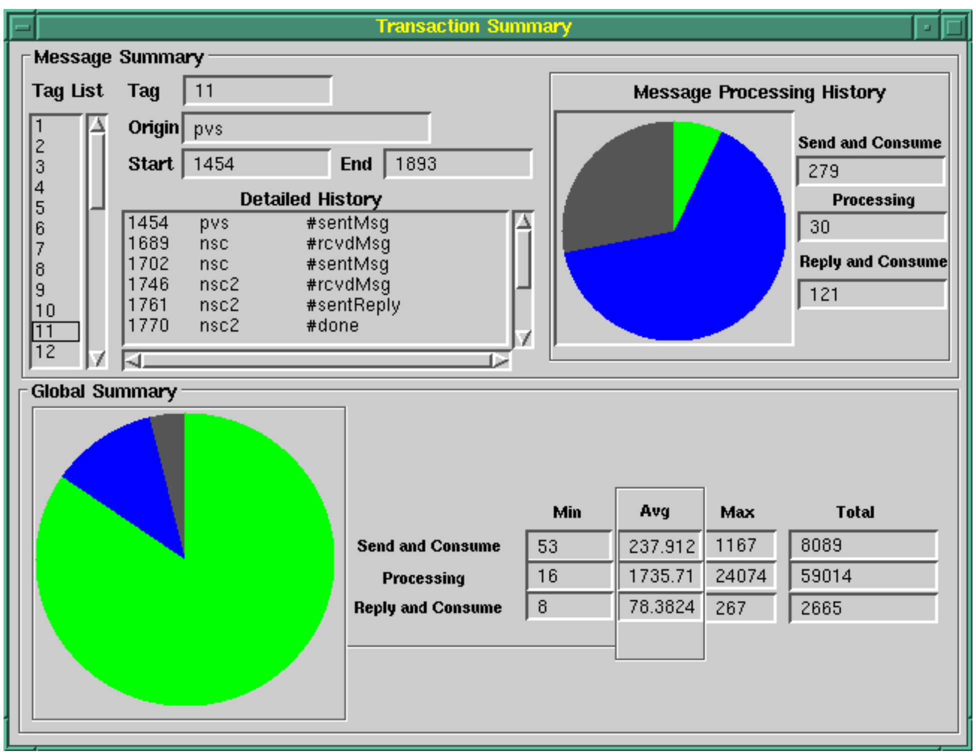

Figure 6. The Transaction Summary View.

Figure 7 shows an example of the Annotation View. The view indicates where the first speedup occurred. The user can select performance events of interest, and the system will annotate the time-line to indicate where the events occurred. Each of these events can be annotated with a comment indicating the probable cause of the event and an indication of where in the user's code the performance problem occurred. Thus the user knows exactly where potential problems lie. The situation choices are obtained from a menu that includes such utilization statistics as *register on first speed-up, register on all speed-ups, register on first slow-down, bottle-necks, granularity problems, network flooding,* etc. It also includes such aggregate operations as *asset under-utilized*, *asset over-utilized*, *overloaded message queue* and *excessive message processing time*.

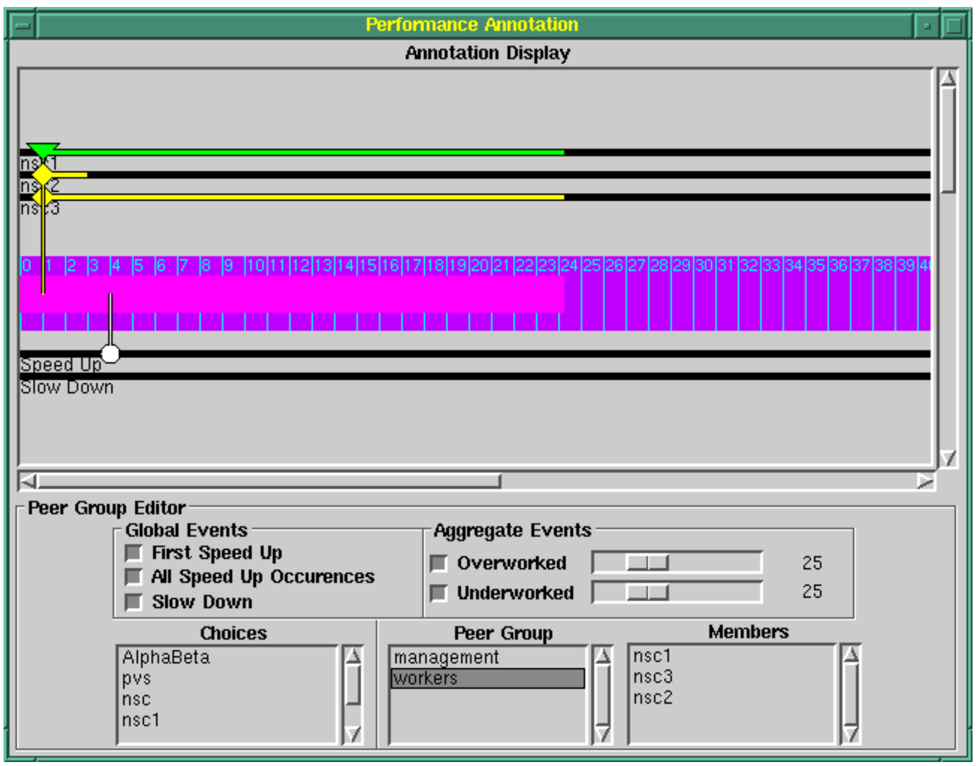

Figure 7. The Performance Annotation View .

The nature of the model and the accessibility of compiler information allow us to provide meaningful annotations to the user’s code. In this way, the tool can offer the user some insight as to where the performance error is occurring.

4-CONCLUSIONS

Debugging distributed programs should not be treated in isolation from the model and system in which the programming takes place. If a utopian system and model could guarantee that no errors could occur in a program, the best debugger would be no debugger at all. Unfortunately, in the real world this is not the case. The best we can do is prevent or minimize certain types of errors and then provide debugging tools uniquely suited to the programming environment and the types of errors that are most likely to occur. This paper has shown that template-based distributed computing systems are able to eliminate or minimize a significant number of programmer errors directly associated with the distributed domain. In particular, much of the tedious error-prone coding involved with establishing communication links, packing and unpacking messages, managing inter-process communication and handling synchronization can be automated. The result is error-reduced code and greater programmer productivity. Enterprise implements a template-based model of distributed computing within a graphical programming interface, augmented by a collection of integrated tools. Consequently, the transition from sequential programming to distributed computing becomes far less intimidating. The super-computing power of networks of under-utilized workstations is placed within the reach of programmers untutored in the highly specialized skills normally required in distributed computing.

The Enterprise experience also shows that the tools for performance analysis and correctness debugging can be seamlessly and uniformly melded into a user-friendly programming environment. The overhead of learning such a system is greatly reduced through a sort of skill amortization in which an operation or technique learned in one context can be used in another context or generalized to a related operation. By maintaining the same graphical perspective of the program when debugging in the Replay View as when analyzing performance in the Animation View or when designing the program in the Design View, the user can more easily grasp the complexities of the program's behavior and process concurrency. Within such an environment, programmers are less likely to err.

By virtue of its template-based paradigm, Enterprise is able to launch a preemptive attack on errors within the distributed programming domain and shield the user from many of the inherent hazards. These preventive measures combined with customized equipment for debugging in Enterprise will hopefully make the arduous chore of writing and debugging a little less daunting.

Enterprise is publicly available: http://www.cs.ualberta.ca/~enter.

ACKNOWLEDGMENTS

This research has been funded by NSERC and a grant from IBM Canada Limited's Centre for Advanced Studies.

REFERENCES

E. Arjomandi, I. Kalas and W. O'Farrell. Concurrency Abstractions in a C++ Class Library, CASCON'93 Conference Proceedings, Toronto, October 1993.

B. Bacchi, M. Danelutto and S. Pelagatti. Resource Optimization via Structured Parallel Programming. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 13-25, 1994.

H. Bal, M. Kaashoek and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190-205, 1992.

A. Beguelin, J. Dongarra, A. Geist, R. Manchek and K. Moore. HeNCE: A Heterogeneous Network Computing Environment, Carnegie Mellon University, Computing Science Department, Technical Report CS-93-205 (August 1993).

M. Beltrametti, K. Bobey, R. Manson, M. Walker and D. Wilson. PAMS/SPS-2 System Overview. Supercomputing Symposium, pp. 63-71, 1989.

N. Carriero, D. Gelernter, T. Mattson and A. Sherman. The Linda Alternative to Message-passing Systems. *Parallel Computing*, vol. 20, no. 4, pp. 633-655, 1994.

L. Clarke, R. Fletcher, S. Terwin, R. Bruce, A. Smith and S. Chapple. Reuse, Portability and Parallel Libraries. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 171-182, 1994.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, October 1994.

G. Geist and V. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.

A. Goldberg. Concert/C: A Language for Distributed C Programming. IBM T.J. Watson Research Center, Yorktown Heights, New York, 1993.

A. Grimshaw, W.T. Strayer and P. Narayan. Dynamic Object-Oriented Parallel Processing. *IEEE Parallel and Distributed Technology*, vol. 1, no. 2, pp. 33-27, 1993.

A.R. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, 1985.

P. Iglinski. An Execution Replay Facility and Event-Based Debugger for the Enterprise Parallel Programming System, M.Sc. Thesis, University of Alberta, 1994.

G. Lobe, D. Szafron and J. Schaeffer. The Enterprise User Interface. TOOLS (Technology of Object-Oriented Languages and Systems) 11, R. Ege, M. Singh and B. Mayer (editors), pp. 215-229, 1993.

P. Mehrota and T. Pratt. Language Concepts for Distributed Processing of Large Arrays. Principles of Distributed Computing, pp. 19-28, 1982.

J. Schaeffer, D. Szafron, G. Lobe and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, vol. 1, no. 3, pp. 85-96, 1993.

Z. Segall and L. Rudolph. Pie (A Programming and Instrumentation Environment for Parallel Processing), *IEEE Software*, vol. 2, no. 6, pp. 22-37, 1985.

S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons and R. Title. A Scalable Debugger for Massively Parallel Message-Passing Programs, IEEE Parallel & Distributed Technology, Vol. 2, No. 2 (Summer 1994), pp. 50-56.

Sun Microsystems. Remote Procedure Call Programming Guide. Sun Microsystems, G. Lobe, D. 1986.

D. Szafron and J. Schaeffer, An Experiment to Measure the Usability of Parallel Programming Systems. To appear in Concurrency Practice and Experience, 1996.

D. Woloschuk, P. Iglinski, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer and D. Szafron. Performance Debugging in the Enterprise Parallel Programming System, CASCON'95 CDRom Conference Proceedings, Toronto, November 1995.