

## **Implementation Techniques for a Parallel Relative Debugger**

D. Abramson  
R. Sasic  
G. Watson §

School of Computing and Information Technology  
Griffith University  
Kessels Rd, Brisbane,  
Queensland, 4111  
{davida, sasic}@cit.gu.edu.au  
Phone: +61-7-875 5049  
Fax: +61-7-875 5051

§ Division of Information Services,  
Griffith University.  
g.watson@ins.gu.edu.au

### **Abstract**

*This paper discusses a new debugging strategy for parallel programs, called parallel relative debugging. Relative debugging allows a user to compare the execution of one program to another, and this can be used to trace errors. This technique has been found to significantly aid in problem determination. A prototype sequential relative debugger, called Guard, has already been constructed and has been used in a number of real world situations. However, the control logic it uses is not sufficiently powerful to support the debugging of parallel applications. In this paper we describe how dataflow can be used to provide a very rich control mechanism that is well suited to the parallel environment. We illustrate the system by a worked example.*

### **1. INTRODUCTION**

The development of massively parallel computer architectures and languages has created a serious problem for the software industry. Many applications demand the high performance which can be achieved through parallel systems, however, porting code to these machines can be extremely difficult. The high cost of software development means that it is desirable to make use of as much of the existing code as possible. However, this poses a number of problems. For example, a program may need to be translated to another language, and may execute on a computer architecture which is radically different from the one on which it was originally developed. Many parallel systems require specific algorithmic changes in order to exploit their performance advantages, and these may require substantial changes to

the original source code. The variation in machine architectures means that the code may need to be implemented using a shared memory, message passing or data parallel paradigms, or a combination of the three. At present, it is difficult to maintain one source program across all these paradigms, thus there is the need to build, debug and maintain many different versions.

Debugging parallel programs is also a difficult task. There has been some progress in developing debugging tools which understand the management of many co-operating processes [6, 8, 9, 11, 12, 13, 15, 16, 18, 19, 20, 21, 24]. All of these tools are based on the premise that the programmer understands the detailed operation of the code and that it is possible to detect errors by comparing the behaviour of the program with the behaviour expected by the programmer. Whilst this approach is effective, it has a number of serious drawbacks. First, the programmer who is porting the code may not be the author of the original version. Thus, they may not have a good understanding of all of the algorithms and data structures. Second, it relies on the programmer to formulate an expected model of execution, which is difficult for large complex programs.

The traditional approach to debugging a program which produces different (possibly incorrect) output on two platforms is somewhat laborious. A user may invoke a conventional debugger and examine the new code in isolation, effectively repeating the debugging phase which occurred when the program was written. This approach has many inherent problems. For example, it may be some time since the code was written, and thus the programmer may not recall all of the details of the program. Further, the person performing the port may not be the original programmer, and thus may not be familiar with all of the details of the code. This situation also arises when the code has been built by a team. Another

approach is to instrument both the new code, and the original one with output diagnostics, and then to perform either a manual or automatic comparison of data values. This approach is extremely time consuming and error prone when the codes manipulate large amounts of data. If the error does not occur quickly, it may be necessary to invoke many debugger commands manually.

In 1993 we introduced the concept of a *relative debugger*,<sup>1</sup> a tool which assists a user in debugging a program by using another one as a reference version [1, 3, 4, 5]. Relative debugging differs from conventional debugging because the user need not be concerned with the exact state of a process in absolute terms, but simply relative to another program. For example, rather than examining each item of a large floating point array, and deciding whether the values are *correct*, the user *compares* the array in one program with its equivalent in the reference code. The result of the comparison allows the user to determine whether the array is correct without knowing the exact data values. There are two ways of comparing data, and these are discussed in detail in Section 2. First, it is possible to use *declarative* (a priori) specifications that certain data items must be equivalent. Second, an *imperative* comparison can be invoked by a user on demand. These two modes provide considerable flexibility and dramatically simplify the process of debugging.

This paper discusses the issues in implementing the control logic of a relative debugger. We show how a relatively simple method is adequate for most sequential programs, and discuss its implementation in a prototype called Guard. However, this simple mechanism does not handle the complexities of arbitrary parallel codes. We then show how a dataflow execution mechanism can be used to control parallel programs.

## 2. GUARD - A RELATIVE DEBUGGER

Guard is a sequential relative debugger developed at Griffith University in Brisbane Australia. Guard uses a client server architecture to allow the user to execute the debugger and the two applications on three different platforms, as shown in Figure 1. Guard behaves as a client and interacts with the user. Each of the applications is controlled by a debug server, which then communicates with the client by a remote procedure call. This separation of function means that Guard can be written in a machine independent way and machine specific information is contained in the servers.

When debugging a program, the user first determines whether the contents of various data structures are incorrect, by comparing the data values at key points in the reference code and the one being debugged. The user then interactively refines the region being examined until the cause is located. We have used this techniques in a number of real world case studies, including a Global

Climate Model (GCM) [1] and a scientific pollution modelling code [17].

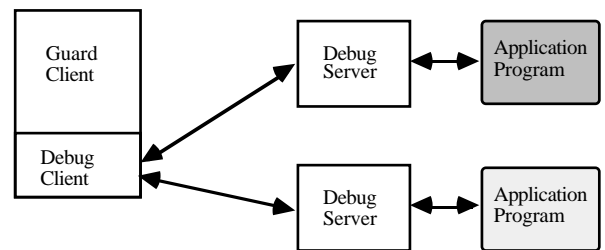


Figure 1 - Client Server Architecture of Guard

Guard provides two ways of comparing data. *Assertions* are a priori statements provided by the programmer about which data structures should be equivalent and at which points in the execution of the code. Because it is difficult to provide good temporal specifications which define when data structures should be equivalent, Guard uses line numbers. Assertions are implemented by planting breakpoints at all of the specified lines in each of the applications. When the breakpoints are triggered, Guard extracts the data from each process and performs comparison automatically. The form of an assertion is as follows:

```
assert process1::variable1:file1@line1 =
       process2::variable2:file2@line2
```

This assertion states that variable1 in process 1 (which will have been started on a particular machine) at line1 in file1, should be the same as variable2 in process2 at line2 in file2. It is the responsibility of the debugger to check the assertion each time the two lines are encountered. The mechanism for performing this will be discussed later in the paper.

*Imperative comparisons* are statements which can be issued by the programmer when the two programs are suspended. Typically, the user plants a number of breakpoints in each program and then executes them until they are triggered. It is then possible to compare any data structure between the two programs, or even within the same program. A useful combination of assertions and imperative comparisons consists of waiting for an assertion to fail, and then manually issuing a number of comparison statements to determine the cause of the violation. Imperative comparisons have the following form:

```
compare process1::variable1 process2::variable2
```

Differences in data structures can be reported through a number of techniques. First, values of scalars and small arrays can be written to standard output in text format. Second, Guard provides a basic graphics facility which can display differences in two dimensional arrays using a pixel map. Black pixels are used to denote which elements are different and white pixels are used to show which elements are the same. This mode is useful for detecting patterns in the data. For example, if a loop

<sup>1</sup> Patent Pending

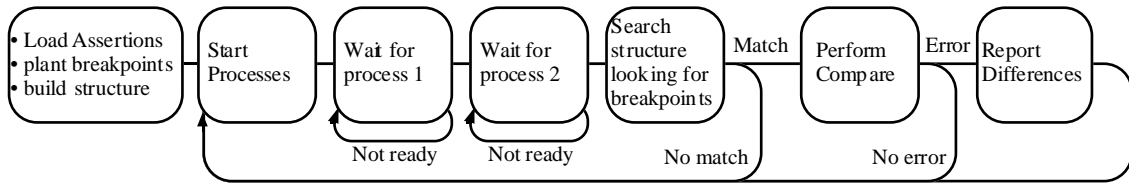


Figure 2 - Control Flow for sequential debugger

terminates early, then a line of pixels is visible in the error report. Finally, the difference in the data can be exported to an external scientific visualisation package. We have experimented with a number of displays using IBM's DX product.

Tolerances are used for inexact equality of floating point number. Thus, it is possible to set a tolerance below which errors are not reported. Given the variation in floating point behaviour on different platforms this mode can be used to detect errors only when the difference becomes significant.

In order to compare data structures, they should either be type conformant or amenable to type coercion. Further, arrays should be of the same shape and size. Guard provides a sub-array syntax for comparing sub sections of larger arrays; this is particularly useful when comparing parallel numeric programs with their sequential counterparts because the parallel arrays may contain extra rows and columns for supporting the parallel algorithm. Guard also provides an index permutation function, which means it is possible to compare arrays in which the indexes are ordered differently. Guard allows the two programs to be written in different languages, and understands the way in which various languages layout their data. We are currently producing a version of Guard which understands the data layout rules of an experimental data parallel language called ZPL [14].

### 3. CONTROL MECHANISM FOR SEQUENTIAL PROGRAMS

Imperative comparisons are similar in nature to the commands found in conventional debuggers like gdb [25]. They are executed immediately after being issued and use the *current* state of the processes at that time. Thus, the compare statement is relatively easy to implement because it can use existing debugger infrastructure. It must be able to extract the data from two processes and perform a difference operator on the data structures. The ability to perform this comparison in a heterogeneous distributed environment is supported by the client-server nature of Guard.

However, the assertions described in the previous section require more support by the debugger because they are specified declaratively before the codes are executed. Consequently, they do not act immediately on the executing processes. To implement assertions the debugger must:

- Save information for each assertion, which includes two sets of data structure names and breakpoint information
- Set breakpoints in both programs at the appropriate locations
- Execute both programs concurrently, possibly on different platforms
- Determine when to *fire* assertions
- Extract the data from the two programs
- Perform a comparison

In other words, assertions can be thought of as triggers for imperative compare statements. In this paper we address the issue of the control logic which is necessary to fire assertions. Assertions must be saved in a structure prior to program execution, and must be evaluated when breakpoints are encountered. When both the reference program and the code being debugged are sequential, a relatively simple control mechanism can be used for detecting which assertions are ready, as shown in Figure 2.

In this scheme, the debugger executes a sequential match process. Both processes are started on their respective machines, and then the debugger waits for a breakpoint from the first process. When the breakpoint is encountered, the debugger saves the current breakpoint address and waits for the second process. Only once it has a pair of breakpoint addresses does it search the assertion list. If it finds a match, then it performs the associated comparison. If the pair of addresses does not find a match in the assertion structure then both processes are resumed without a comparison. In this case, the processes clearly executed a path which was not expected, and so it is not sensible to perform any comparison. In this case the debugger reports the anomaly and continues execution. Likewise, if either program fails to meet a breakpoint at all then the debugger will not perform any comparisons.

The scheme described above has been implemented in Guard and has been used for a number of trials. It is effective for handling a number of unrelated assertions which have different breakpoint addresses. It can also support assertions which share breakpoint addresses, providing they all have the same addresses in both processes. However, if two assertions share only one breakpoint, then the logic fails to match correctly, because the data from one process is required for two different comparisons. Further, it requires a significant

modification if the debugger is used for parallel programs, which will be discussed in the next section.

## 4. DEBUGGING PARALLEL PROGRAMS

### 4.1 Nature of parallel programs

As discussed in section 3, the current sequential matching logic is adequate when both programs are sequential, providing there are certain restrictions on the way assertions share breakpoint addresses. However, in practice this can be restrictive when it is necessary to compare a data structure from one point of the reference code with two different places in the code being debugged.

More importantly, parallel programs do not adhere to strict sequential control patterns. Consider the comparison of a sequential reference code (*seq*) with a parallel implementation which uses two slaves (*p1* & *p2*), as shown in Figure 3. The sequential code maintains an array A[100], which is decomposed across the two slaves into two arrays, B[50]. Accordingly, sections of the array A in the reference code must be compared with each of the B arrays from the slaves at line 100, using the following two assertions:

```
assert seq::A[0..49]:seq.c@100 =
      p1::B:par.c@100
```

```
assert seq::A[50..99]:seq.c@100 =
      p2::B:par.c@100
```

However, if these two assertions are treated independently, then they will fire at different times. Even worse, the first assertion will capture A when *seq* reaches line 100, and then will restart both *seq* and *p1*. Thus, the second assertion will only capture A when *seq* reaches line 100 for a second time. In order to realise the correct semantics, it is necessary to indicate that the two should behave as a single assertion, and the data structure A should be captured in the process *seq* at the same time. An enhancement of the following description language achieves this requirement, by specifying that the comparison should be performed with the same instance of A:

```
assert (seq::A[0..49]:seq.c@100 =
      p1::B:par.c@100) &&
      (seq::A[50..99]:seq.c@100 =
      p2::B:par.c@100)
```

However, it is clear that the simple control logic discussed in section 3 is unable to support the required semantics, because the array A must be captured in one atomic operation, and then used in two subsequent comparisons. But the two processes *p1* and *p2* may arrive at their breakpoints out of synchronisation, and thus A must be saved until it has been used in both comparisons. In section 4.2 we describe a mechanism which can support the required semantics.

### 4.2. A Parallel Control Mechanism

The discussion to date has highlighted that in order to support parallel relative debugging, the control mechanism must have the following attributes:

- It should be possible to extract data from one process independent of another.
- Breakpoints must be handled any order regardless of the static structure of the assertions.
- It should be possible to reuse a data item in many different assertions
- The control mechanism needs to be multi-threaded to manage many processes at once.

It is possible to provide all of these attributes by using a dataflow execution mechanism [7, 10]. In a dataflow environment, nodes (or functions) only execute when they have all of their operands. The *assertion* statement can be considered as a function which compares data only when it has data from two processes. Moreover, the data can arrive on the two inputs at any time, and if it is present on one input only it must be stored until the data arrives on the other input. Likewise, if a data item is required in more than one assertion from the same breakpoint, then it must be duplicated for each assertion. Accordingly, the assertions can be thought of as a dataflow graph, where the nodes perform functions and the arcs transmit tokens of data, captured at various instances of execution.

Using the example discussed in the previous section, Figure 4 shows a sample graph for the following assertion list:

```
assert seq::A[0..49]:seq.c@100 = p1::B:par.c@100
```

```
assert seq::A[50..99]:seq.c@100 =
      p1::B:par.c@100
```

```
assert seq::A[0..9]:seq.c@150 =
      p1::B[0..9]:par.c@90
```

```
assert seq::A[50..59]:seq.c@150 =
      p2::B[0..9]:par.c@95
```

The graph contains 4 compare nodes, each of which tests for a difference in their inputs. If there is a difference then this is reported through the node output. Figure 4 shows tokens moving along the input arcs, arriving at different times. Only when both tokens are present does the node fire. Tokens are produced as a result of breakpoints in each of the processes.

A few debuggers have used dataflow as a control mechanism in the past [15, 20]. In [20] this was concerned with using the breakpoint as an event trigger for a piece of debugger code. Likewise, TV [8] allows the user to write expressions which are executed when a breakpoint fires, and these can be considered like dataflow expressions. Our work extends the previous work by building complex tokens containing data from the processes being traced, and then using this data in the

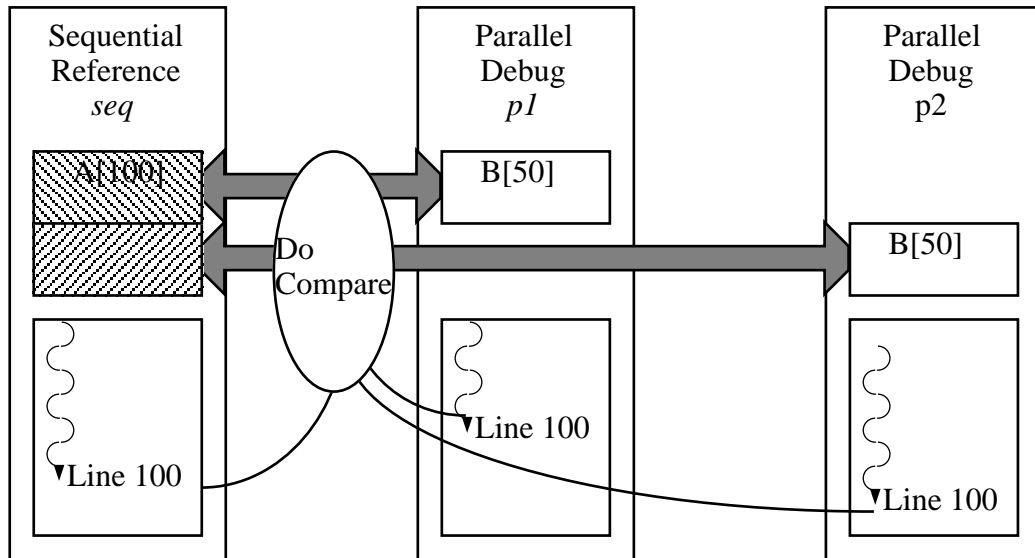


Figure 3 - Sequential Reference compared with Parallel version.

debug statements. The dataflow model makes it possible to detect when both operands of a compare have arrived.

In the next section we discuss some of the implementation considerations.

model. When developing the implementation of such a model, we had to address a number of issues, including:

- the type of dataflow model to use;
- the types of nodes in the graph;
- what information will be carried by tokens in the graph;
- how the graph and tokens should be represented internally; and
- the representation of data in a heterogeneous environment.

These issues are addressed in more detail in the following sections.

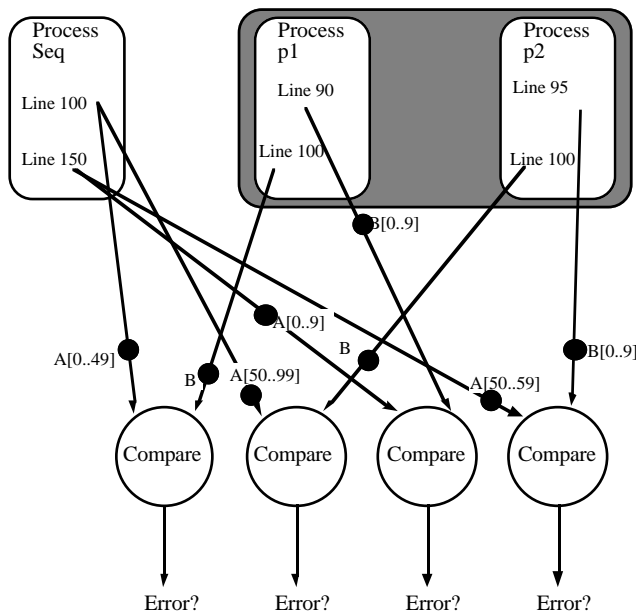


Figure 4 - Dataflow graph representing assertions

## 5. IMPLEMENTATION ISSUES

As discussed in the previous section, the most appropriate control mechanism to suit debugging in a parallel environment is one that utilizes a dataflow

### 5.1 Type of Dataflow Model

Dataflow architectures can be classified into two main categories: static and dynamic. In addition, there have been implementations of hybrid schemes [7, 10]. The primary difference between the static and dynamic models is the number of tokens that may be present on an input at one time. The static model allows only one, while in the dynamic model, many tokens may be present on the input to a node. Hybrid models are a combination of both strategies [2].

In our implementation, we chose to use a variation of the static model in which tokens form a FIFO queue on the input arcs. In this model tokens are consumed from the head of each queue. This makes it possible for us to restart a process immediately after the data has been extracted rather than wait for it to be processed. Each node also contains information which specifies which of the following conditions must be satisfied before the node will fire:

- Fire only when tokens exist on all inputs;
- Fire when a token appears on any input;
- Fire once when the graph is loaded.

A node that fires will always consume its input tokens, but may not generate an output token. Nodes that have more than one output automatically replicate any output token on the different outputs. There are no limits on the number of inputs and outputs arcs for each node.

## 5.2 Graph Implementation

Internally, the dataflow graph is represented as a linked list of nodes. The data structure representing each node contains information about the number of inputs and outputs, the function that the node is to perform, and various other items.

Nodes that have tokens present on their inputs are linked into a list of *active* nodes. Execution of the graph consists of scanning this list and checking the inputs on each node. When the number of tokens matches the number required to fire the node (taking into account the class of the node), then the node is fired and the associated node function executed.

Since there are no tokens in the graph at the start, some mechanism must be used to generate some initial tokens. The generation of these initial tokens is handled by class (iii) nodes, which are primed with static information such as a program name.

## 5.3 Token Implementation

Creation and destruction of tokens involves the allocation and deallocation of memory resources, a potentially time consuming process. For efficiency, our implementation allocates a pool of tokens when it is first started, and can expand the pool dynamically if required. A bitmap structure is used to mark tokens in the pool that are currently free. The process of allocation then becomes simply a matter of finding the first free location in the bitmap, and then returning a reference to the corresponding token. Deallocation just requires the appropriate bit in the map to be set. Token duplication is managed by maintaining a reference count on the token.

Tokens are statically typed using a type tag. Nodes accept tokens of generic type by default. However, node functions can impose strict typing on the input tokens if required.

## 5.4 Data Representation

Central to the concept of relative debugging is the comparison of data from programs executing in a heterogenous environment. While it is relatively easy to perform conversions between simple data types such as integers and characters, it becomes more problematic for floating point values and complex data types such as arrays and structures. In order to overcome the problems inherent in this approach, it is important that an architecture independent format (AIF) for data representation is used.

All manipulation of data by the debugger is carried out in the AIF, using AIF support routines. Routines are also supplied to facilitate the conversion between the local machine dependent data format and the AIF.

Data represented in AIF consists of the components: the actual data formatted as a sequence of bytes of variable length; a format descriptor string (FDS) that consists of a variable length sequence of ASCII characters that describes the layout of the data; and the machine specific formatting information about the data.. The AIF support routines generally require both these components for operation.

The AIF makes some assumptions, including:

- 1) floating point values use IEEE implementation;
- 2) byte ordering is big endian (conversion carried out if required); and
- 3) all simple data types (including pointers) will fit into a maximum of 8 bytes.

The FDS can be parsed to determine the exact nature of the variable, including its type and structure. The syntax is sufficiently powerful describe both simple and complex types.

Below are some sample FDS strings for a few variable types:

Declaration	FDS	Meaning
<code>char c;</code>	"cs"	signed char (1-byte integer)
<code>int i;</code>	"is4"	4-byte signed integer
<code>float f[100];</code>	"[r0..99is4]f4"	array of 100 floating point numbers
<code>struct { int f1; float f4 };</code>	"{8=f1@0#32:is4,f2@32#32:f4}"	structure containing two fields, a 4-byte integer and a 4-byte floating point number

In these examples, a character is described by the string "cs", a four byte integer by the string "is4". A floating point array requires a description of both the index range, the type of the indexes and the type of the array elements, and is thus built from a range from 0 to 99, an index type of 4 byte integers and a base type of 4 byte floating point numbers. A structure with an integer and a floating point number is more complex. It contains a description of the size of each of the items (in bits) and the starting offset in the struct (also in bits), plus the base type of each of the items. Using this syntax we have been

able to represent all of the types found in major programming languages, including user defined types.

## 5.5 Debug Interface

A high level debugging interface is used to control the application programs and also to extract their state for comparison. This is provided by the Dynascope debug library [22, 23], which provides a client-server structures interface. Dynascope provides function to control the execution of programs on arbitrary machines, extract symbol table information and read and write variables in the process address space. More details of Dynascope can be found in [22] and [23].

## 6. A WORKED EXAMPLE

In this section we illustrate the use of dataflow graphs with a small, but realistic, example. Figure 5 shows a graph which performs the following assertion between two sequential programs, "prog1" and "prog2":

```
assert prog1::var1@prog1.c:8 =  
      prog2::var1@prog2.c:10
```

The graph begins by invoking both programs and planting breakpoints at line 8 in prog1 and line 10 in prog2. The *Set\_Breakpoint* node returns a token which is then used to fire a *Continue* node. *Continue* starts the program running and waits for a breakpoint to be encountered, and then it emits a token. The output of the *Continue* node causes the variable "var1" to be read from both programs, and the data is then compared using a *Compare* node. If the data is equivalent, then the programs are both restarted, however, if an error is detected then the differences are reported to the user via a *Display* node.

A number of *Constant* nodes are used to inject literals into the graph. Some of these only fire once when the graph is started, whilst others are re-triggered repeatedly.

It is clear from this example that the system is not sensitive to the exact timing or order of breakpoint events. Further, additional assertions can be integrated easily into the graph, again without regard to timing issues.

## 7. CONCLUSION

In this paper we have described the design and implementation of an elegant and effective control mechanism for a relative debugger based on a dataflow model. Dataflow was chosen to provide support for the debugging of parallel codes relative to serial codes, where it can not be guaranteed information will arrive in a serial manner. The model also allows increased flexibility in the types of data comparisons that can be performed. The design is further enhanced by allowing additional functionality to be easily and quickly implemented and tested, if required.

We have implemented a prototype of the new control mechanism, which has been successfully trialed on a

number of sequential programs. This has demonstrated that the new control mechanism can effectively replace the one used in the current implementation, and has addressed the primary issue of how the mechanism will interact with the application programs.

The prototype currently accepts dataflow graphs written in a simple linear syntax only. Plans are underway to examine alternate methods of specifying the dataflow graphs, including a linear declarative text form (similar to the current assertion syntax) and also using a visual graph editor. Possible additional enhancements include the ability to combine graph nodes into modules in order to reduce the overall complexity of the dataflow graphs.

In this paper we have not discussed the details of how parallel codes can be debugged using our relative paradigm, nor issues such as how the debugger will interact with parallel task managers. Rather, the paper has concentrated only on the core algorithms required for data gathering and comparison. We are currently in the process of designing nodes, in addition to the basic node set, that will provide support for parallel codes. We will report our experiences in addressing these issues, and present a more complete parallel relative debugger in a future paper.

## Acknowledgments

This work is supported by the Australian Research Council. We wish to acknowledge our collaborators, Prof Larry Snyder, Dr Ian Foster and Professor Geoff Dromey who proof read drafts of this paper. We also thank Mr Borut Lesjak who designed and implemented the Architecture Independent Format (AIF) used in the implementation, plus sections of the Dynascope debugging system.

## References

- [1] Abramson D., Foster, I., Michalakes, J. and Susic R., "Relative Debugging and its Application to the Development of Large Numerical Models", IEEE Supercomputing 1995, San Diego, December 95.
- [2] Abramson, D and Egan, G, "The RMIT Dataflow Computer: A Hybrid Architecture", The Computer Journal, Vol 33, No 3, June 1990, pp 230 - 240.
- [3] Abramson, D.A. and Susic, R. "A Debugging Tool for Software Evolution", CASE-95, 7th International Workshop on Computer-Aided Software Engineering, Toronto, Ontario, Canada, July 1995, pp 206 - 214.
- [4] Abramson, D.A. and Susic, R. "Relative Debugging using Multiple Program Versions", Key Note Address, 8th Int. Symp. on Languages for Intensional Programming, Sydney, 3-5th May,

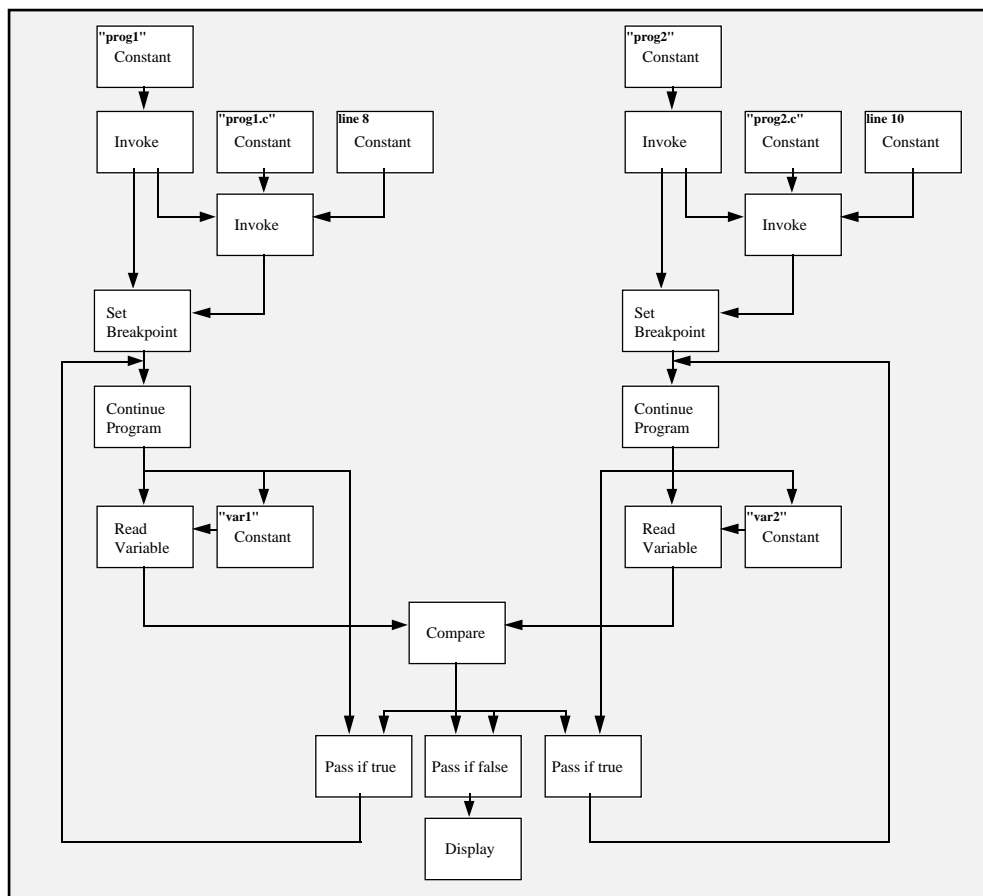


Figure 5 - Sample Dataflow graph

1995. In *Intensional Programming I*, World Scientific, ISBN 981 - 02 - 2400 - 1.

- [5] Abramson, D.A., and Sosic, R. "Method for Testing and Debugging Computer Programs", Australian Provisional Patent, PM5196/94.
- [6] Aral, Z., Gertner, I. and Schaffer, G. "Efficient debugging primitives for multiprocessors", Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pages 87-95. ACM, 1989.
- [7] Arvind, Bic, L. and Ungerer, T. "Evolution of Data-flow Computers", Chapter 1, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.
- [8] BBN Corporation, "The Total View Debugger Reference Manual"
- [9] Cheng, D. and Hood, R. "A Portable Debugger for Parallel and Distributed Programs", Proceedings of Supercomputing 94, Dec 94.
- [10] Dennis, J. "The Evolution of "Static" Data-flow Architectures", Chapter 2, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.
- [11] Elshoff, I. J. P., "A distributed debugger for Amoeba", Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pages 1-10. ACM, 1988.
- [12] Griffin, J. H, Wasserman, H. J. and McGavran, L. P. "A debugger for parallel processes", *Software-Practice and Experience*, 18(12):1179-1190, December, 1988.
- [13] LeBlanc, T. J. and Mellow-Crummey, J. M., "Debugging parallel programs with instant replay", *IEEE Transactions on Computers*, 36(4):471-482, April 1987.
- [14] Lin, C. and Snyder, L. , ZPL: an array sublanguage. In U. Banerjee, D. Gelernter, A. Nicolau, And D. Padua (Eds.) *Languages and*

- Compilers for Parallel Computing, Springer-Verlag, pp. 96-114, 1993.
- [15] Lumpp, J, Casavant, T., Siegel, H.J., and Marinescu, D. "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", Proceedings of 10th Intl. Conference on Distributed Computing Systems, IEEE, 1990, pp 476 - 483.
- [16] McDowell, C. E. and Helmbold, D. P. , "Debugging concurrent programs", ACM Computing Surveys, 21(4):593-622, December 1989.
- [17] McRae G.J., Russell A.G. and Harley R.A., 1992. CIT photochemical airshed model -users manuals, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.
- [18] Miller, B. P. and Choi, J.-D., "A mechanism for efficient debugging of parallel programs", Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation, pages 135-144. ACM, 1988.
- [19] Moher, T. G. "PROVIDE: A process visualization and debugging environment", IEEE Transactions on Software Engineering, 14(6):849-857, June 1988.
- [20] Olsson, R. A. , Crawford, R. H. and Ho W. W. . "A dataflow approach to event-based debugging", Software-Practice and Experience, 21(2):209-229, February 1991.
- [21] Ramsey, N. and Hanson, D.R. "A retargetable debugger"., Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, pages 22-31. ACM, 1992.
- [22] Sobic, R. "A Procedural Interface for Program Directing", Software Practice and Experience, Vol 25, No 7, July, 1994, pp 767 - 787.
- [23] Sobic, R. "Design and Implementation of Dynascope, a Directing Platform for Compiled Programs", Vol 8, No 2, Spring, 1995, pp 107 - 134.
- [24] Satterthwaite. E. " Debugging tools for high level languages", Software-Practice and Experience, 2(3):197-217, July-September 1972.
- [25] Stallman, R. M., GDB Manual, Free Software Foundation, Cambridge, MA.