Debugging Tools for Concurrent Logic Programming

T. CONLON¹ AND S. GREGORY²

¹ Department of Mathematics and Computing, Moray House College, Edinburgh EH8 8AQ
² Department of Computer Science, University of Bristol, Bristol BS8 1TR

A problem which confronts the developers of concurrent logic programming (CLP) systems concerns the design of the programming environment, particularly the provision of debugging tools. Debugging tools are useful for many activities besides identifying bugs: they can help in program testing and demonstration, in software experiments, and in teaching the language semantics. For CLP languages the questions of what debugging tools should be provided, and of how they should be used, are still open. Although the languages are closely related to other logic programming and concurrent programming languages, they are sufficiently different that new debugging techniques are required.

This paper describes a primarily channel-oriented debugging methodology and a set of debugging tools that we have developed in the light of our experience in using and teaching the CLP language Parlog. With these tools a programmer can test a program by observing communication on channels, opening up a process to examine the activity on internal channels and, if necessary, to check in detail the execution steps of a process.

Almost all of the ideas, which are equally applicable to other CLP languages, have been implemented in two commercial Parlog systems. As well as fulfilling the needs of existing users of CLP languages, we believe that the tools emphasise the attractions of CLP as a concurrent programming paradigm, since they are made possible by the unique attributes of CLP.

Received November 1990, revised February 1991

1. INTRODUCTION

Concurrent programming and logic programming have both been the focus of considerable research during the past decade. The reasons are clear: concurrent programming is necessary to support a growing number of advanced applications and to exploit the emerging multiprocessor hardware architectures; logic programming attempts to root the activity of software development in formal logic, and hence to make software development more reliable, productive, and better matched to the increasing number of artificial intelligence-oriented projects.

Particularly interesting and, we think, fruitful has been research into the intersection between these subjects. From this research has emerged a new family of languages: the concurrent logic programming (CLP) languages, exemplified by Parlog, Concurrent Prolog, GHC, and Strand. The CLP languages attempt to combine the benefits of both parent disciplines into a coherent whole. Some of them are now available commercially and it seems likely that they will become more widely used in the near future.

One of the problems which confronts the developers of CLP systems concerns the design of the programming environment, including (the subject of this paper) the provision of debugging tools. Although debugging tools represent only a part of the programming environment, there are several reasons why they are of special interest:

1. Existing CLP systems have poor debugging tools. Yet experience of other programming systems suggests that programmers are more likely to be receptive to, and productive in, a new system if good debugging tools are available.
2. Effective debugging tools need to be tailored to the features of the target language. For CLP languages the questions of what debugging tools should be provided, and how they should be used, are still open. It seems likely that some of the notions developed for sequential logic programming (notably Prolog) will apply, as will some ideas drawn from imperative concurrent languages (such as Occam). But, as we explain below, the CLP languages are sufficiently different from both of these types of language that new debugging techniques are required.
3. Procedural (as opposed to static or declarative, for example) debugging tools are useful for many activities besides identifying bugs: they can help in program testing and demonstration, in software experiments, and (what is particularly important for a new programming paradigm) in teaching the language semantics.

In this paper we describe a set of debugging tools that we have developed in the light of our experience in using and teaching the CLP language Parlog, though the ideas are equally applicable to other CLP languages. Almost all of the tools that we propose have been implemented in two commercial Parlog systems, MacParlog and PC-Parlog. As well as fulfilling the needs of existing users of CLP languages, as noted above, we believe that the tools emphasise the attractions of CLP as a concurrent programming paradigm, since they are made possible by the unique attributes of CLP.

The next section of the paper introduces the main concepts of Parlog and compares it with the related languages Prolog and Occam. In Section 3 we discuss debugging methodology and propose a mainly channel-oriented approach to concurrent logic program debugging. We suggest that three kinds of tool are necessary to support different views of program execution. Sections 4–7 explain the principles behind these tools and provide some illustrations of their implementation and use. The final section summarises our debugging proposals and describes some related and future research.
2. CONCURRENT LOGIC PROGRAMMING

First we provide a brief introduction to concurrent logic programming through Parlog. For a more detailed treatment of Parlog the reader is referred to refs 7 and 11; other references on concurrent logic programming include refs 6 and 17.

2.1. Declarative interpretation of programs

A program in Parlog is a set of procedures, each defining one relation. As an example, Fig. 1 shows a Parlog definition of the qsort relation, which implements the Quicksort algorithm. To those acquainted with Prolog the Parlog version will be quite recognisable. The only apparent novel feature is the mode declaration, which here serves to specify the first argument of qsort as input and the second as output. Mode declarations are significant to the operational interpretation of programs, which we describe below.

mode qsort (?,:),
qsort ([], []). qsort ([N|Rest], Sorted) <= partition (N, Rest, LessN, MoreN),
qsort (LessN, SortedLess),
qsort (MoreN, SortedMore),
append (SortedLess, [N|SortedMore], Sorted).

Figure 1. Parlog procedure for Quicksort.

In the declarative interpretation of programs, however, Parlog procedures can be read in the same way as those written in Prolog: a procedure’s clauses can be understood as descriptive logical statements about the relation which the procedure computes. Thus, the first clause of the qsort procedure can be interpreted as saying:

An empty list sorts to itself.

The second clause can be understood as saying (loosely):

A list containing terms is followed by Rest sorts to a list Sorted if Sorted is the concatenation of two sorted sublists which are obtained by partitioning Rest about N.

The declarative interpretation provides the programmer with a means of checking the program’s correctness without resorting to operational (i.e. behavioural) considerations: a program is correct – in the sense that it will not deliver incorrect results – provided that the conclusion (head) of each clause is logically justified by the clause’s conditions (body).

In concurrent logic programming, as in Prolog, the declarative interpretation is of limited use. For example, the standard definition of the relation merge(X, Y, Z) guarantees that Z will be some interleaving of lists X and Y, but it says nothing about which interleaving could be computed, which may be of crucial importance. Moreover, a program which produces no solution will always be considered correct according to the declarative interpretation. Indeed, in concurrent logic programming, many programs are not even intended to terminate with solution: it is the way in which they interact with the real world that is important. For these reasons, it is essential to consider the operational semantics of the language, described in the next section.

2.2. Operational interpretation of programs

For Parlog, a program’s execution consists of the evaluation of one or more relation calls: each such evaluation is called a process. Thus, the query

\[ \left\langle \left\langle 5, 3, 7, 9, 2, 1, 6 \right\rangle, X \right\rangle, \]

containing a single call to the qsort relation, creates a process which, following the procedure of Fig. 1, computes the solution

\[ X = \{1, 2, 3, 5, 6, 7, 9\} \]

In general, where there are two or more processes, they run concurrently. For example, assuming the definition for randoms shown in Fig. 2, the query

\[ \left\langle \left\langle 50, \text{IntList} \right\rangle, \left\langle \text{qsort(\text{IntList, SortedInts})} \right\rangle \right\rangle \]

would create two concurrent processes. Since the relation calls share a variable the processes can communicate: the integers which are the output of the randoms process provide the input to the qsort process. The execution can be understood in terms of the simple process network shown in Fig. 3. The nodes of the network, annotated by relation names, represent processes and the arcs, annotated by variable names, represent communication channels.

mode randoms (?,:),
randoms (0, []). randoms (N, [Rand|Rands]) <=
N > 0 : random_integer (Rand),
N1 is N-1,
randoms (N1, Rands).

Figure 2. Procedure to generate a list of random integers.

An individual Parlog process has a ‘life cycle’ which is depicted in Fig. 4. The life cycle contains the following events:

Call: This marks the creation of the process, which begins by testing, or input matching, the input arguments of the call against the corresponding head arguments of the procedure’s clauses. The aim is to identify an appropriate, or candidate, clause, and normally the tests with all clauses proceed in parallel. Input matching is a constrained form of unification having the property that only a clause variable may become bound: if a call variable would become bound, input matching suspends. For a call to qsort, the test with the first clause will succeed if the first argument of the call is an empty list; the test with the second clause will succeed if the call argument is a non-empty list pattern; both tests will suspend if the call argument is an unbound variable.
In general, the test in each clause may include a guard evaluation as well as input matching. A guard is an optional initial subset of conditions in the right side of a clause, separated from the clause body by a colon. For example, in Fig. 2, the guard condition \( N > 0 \) will succeed if the first call argument is a positive number, fail if it is non-positive, or suspend if it is an unbound variable.

A clause is a candidate if all of its input matching and guard evaluation has succeeded, or a non-candidate if any part of it has failed; any other clause may become suspended if all input matching and guard evaluation is suspended.

Suspend: If there is at least one suspended clause for a call, and all other clauses are either suspended or non-candidates, the process suspends. It cannot make further progress until some variables in the call's input arguments are instantiated (by another process).

Retry: This marks the reactivation of a previously-suspended process. A process may be suspended and retried many times before eventually reducing or failing.

Reduce: As soon as one of the clauses becomes a candidate, the investigation of alternative clauses is abandoned and the process commits irrevocably to that clause. (If there is more than one candidate clause, one of them is chosen arbitrarily.)

Immediately after commitment, unification is performed between the output arguments in the head of the selected clause and the corresponding arguments of the call. Subprocesses are spawned to evaluate the relation calls contained in the (possibly instantiated) body of the selected clause. The evaluation reduces to these sub-processes in the sense that the parent process succeeds or fails according to the outcome of the child subprocesses.

Succeed: If a process reduces using a clause which contains no output arguments and an empty body, it will immediately succeed. In general, it will succeed only if and when all output unification and all body calls (processes) succeed.

Fail: A process may fail before commitment, if all clauses are non-candidates. After commitment, the process fails if any of the output unification or body calls fails.

2.3. Key features

As a consequence of the above operational semantics, concurrent logic programming is characterised by two attributes: (a) communication channels which are 'first-class' terms in the language, and (b) recursively created processes.

2.3.1. Channels as variables

Perhaps the most unusual feature of concurrent logic programming languages is their method of interprocess communication. A communication channel is a 'first-class' entity in the language: a logical variable, which can be shared by any number of processes. Equivalently, any variable can be viewed as a communication channel. The current term binding for the variable represents a complete history of the communication on that channel. This view has a major effect on our debugging methodology, as we explain in Section 3.

Communication between processes takes place when one of them binds a variable that they share. For example, in query (1) above, the random numbers generated by the randoms process will be made available to qsort incrementally, by a series of bindings such as

\[
\text{IntList} = [37|\text{IntList1}]
= [37,76|\text{IntList2}]
= [37,76,12|\text{IntList3}]
\ldots
\]

with one further integer being produced on each reduction step. This explains the apparent paradox whereby a single-assignment variable can represent a stream of data: the term to which the variable is bound contains a further variable, which can itself be bound, and so on.

In this example, each binding is to a term of the form \([G|V]\) where \(G\) is a ground (variable-free) term and \(V\) is an unbound variable which will later be found by the same process, the producer; this is a very common case, which has a natural interpretation as a stream of messages passing along a channel from the producer to one or more consumers. (The term stream is often used to refer to such a dynamically generated list.)

Much more general forms of communication than this are possible. For example, a non-ground 'message' can be sent, by a binding to a term such as \([t(U)|V]\), where the variable \(U\) can be bound by the producer after the message is sent. More interesting is the case where \(U\) is bound by the consumer process; this represents a 'back communication' from consumer to producer. Moreover, shared variable bindings are not restricted to lists; they can be arbitrary terms, which again can be generated incrementally. Consider a procedure for make_tree, which constructs a tree containing the terms in a given list. Then a query such as

\[\text{<- randoms(50,IntList),}
\text{make_tree(IntList,Tree).}\]

might produce a value for Tree by the following series of bindings:

\[
\begin{align*}
\text{Tree} & = \text{tree(T1,37,empty)} \\
& = \text{tree(tree(empty,76,T2),37,empty)} \\
& = \text{tree(tree(empty,76,tree(T3,12,empty)),37,empty)} \\
& = \ldots
\end{align*}
\]
2.3.2. Recursive process structures

In query (1) above, the qsort process will initially suspend, but as soon as the randoms process has computed for IntList the first binding

\[ \text{IntList} = \{37|\text{IntList}1\} \]

the qsort process will commit to the second clause, unify SortedInts with the head variable Sorted, and spawn four subprocesses, as shown in Fig. 5. Notice how the qsort process, the consumer, is constrained by the supply of data from the randoms producer. The spawned subprocesses will also initially suspend but, as soon as the randoms process sends 76 as the next integer, the partition subprocess will feed this along the MoreN channel enabling one of the qsort subprocesses itself to spawn a further generation of processes.

![Figure 5. Process network for qsort.](image-url)

The idea of 'larger' processes encapsulating 'smaller' ones is fundamental to concurrent logic programming. Analogously, smaller processes can be combined into larger ones. For example, we could define a sort_randoms relation in terms of the randoms and qsort relations (see Fig. 6). The process which evaluates the query

\[ \text{sort_randoms}(50, \text{SortedInts}) \]

(2)

can be viewed abstractly as in Fig. 7; we can ignore the fact that the sort_randoms process actually contains two component concurrent processes identical to those shown in Fig. 3. This 'black-box' view plays an important role in our debugging methodology, as we explain in Section 3.

mode sort_randoms(? , ?).

sort_randoms(N, SortedInts) <-
  randoms (N, IntList),
  qsort (IntList, SortedInts).

Figure 6. Procedure for sort_randoms.

![Figure 6. Procedure for sort_randoms.](image-url)

Figure 7. Process network for query (2).

2.4. Comparison with Prolog and Occam

From the above account, the reader with a knowledge of Prolog will observe that Parlog's execution strategy differs from that of Prolog in several fundamental ways. (In this paper 'Prolog' normally refers to conventional Prolog, as described in ref. 5. Dialects of Prolog extended with some form of co-routining control (e.g. Sicstus Prolog, NU-Prolog, etc.) are closer to Parlog; these will be considered specially in Section 8.3.) Some of the differences most significant to debugging are:

1. Parlog's commitment of any relation call to just one clause means that a Parlog query can only ever compute one solution, whereas Prolog, with its built-in support for search, can compute multiple solutions. It also means that Parlog is free of some of the Prolog features which cause the most difficulty in debugging, notably backtracking, the rescinding of variable bindings, and the 'cut'.

2. In Prolog, the calls of a conjunction are evaluated sequentially, in left-right textual order, so there is only one 'process' active at a time. In Parlog, a conjunction of calls is typically evaluated concurrently, with each call evaluated as an individual process. Obviously, a Prolog call cannot suspend, whereas for Parlog suspension is normal; indefinite suspension (deadlock) is possible, and indeed represents a major hazard for programmers.

3. As a consequence of its sequential execution, a relation call in Prolog never interacts with other sibling calls during its evaluation. Its input data is completely available at the time of the call, while its output is visible to other calls only after the call terminates. An exception is made to enable interactive communication with the real world: side-effect input/output primitives are used.

In Parlog, a call's input data may be produced, and its output consumed, by another concurrently evaluated call. Thus, any process can influence, and be influenced by, the behaviour of another process throughout its lifetime. Input/output therefore requires no additional concepts: the real world is treated just like any other process.

4. Prolog is a deterministic language in the sense that the same program, when run on the same data will always compute the same results. In Parlog this need not be the case. If more than one candidate clause exists for a call, the evaluation must commit to one clause, but the language does not prescribe how the choice should be made. In practical implementations the selected candidate clause is the first to be discovered. Different machines, and indeed the same machine on different runs, could therefore make different choices.

The differences from Prolog relate to Parlog's concurrency. But Parlog also differs markedly from other concurrent languages such as Occam. Processes and channels in Occam are quite different from their Parlog counterparts:

1. In an Occam program, the topology of processes and channels is fixed at compile time. In Parlog, processes can be spawned dynamically and their interconnections can be reconfigured at run time.

2. Parlog channels, since they are actually logical variables shared between relation calls, acquire a value during program execution: the value is a term representing the history of communication on the
channel. An Occam channel is also a communication agency, but it is not a variable and does not accumulate a value.

3. Parlog’s interprocess communication is asynchronous. That is, only input is blocking: a process must suspend until it has sufficient input to commit to a clause. Occam uses synchronous communication – both input and output are blocking: the producer and consumer processes must both be ready before communication can take place.

4. There are several other differences. A Parlog channel is untyped, unbuffered, and can be connected to many processes, any of which can write to it. An Occam channel is strongly typed, buffered, connects exactly two processes, and is unidirectional.

Thus although Parlog combines logic programming with concurrency, it differs from established languages on both sides of its parentage in many ways, some of which certainly affect program debugging.

3. DEBUGGING CONCURRENT LOGIC PROGRAMS

When the behaviour of a program contradicts its specification, a debugging exercise is usually necessary, the first aim of which is to locate the erroneous program component.

Most debugging methodologies view a faulty program as a hierarchy of components which must be systematically searched in order to locate the error. The search can proceed in a top-down or bottom-up manner, or could use some mixture of the two. At each level of the hierarchy each component is tested, preferably in isolation from the rest of the system, initially in ‘black-box’ fashion: judiciously selected inputs are supplied and the outputs are observed and compared to the design specification. If a component is found to be faulty its code is inspected and possibly its execution is traced. If the cause of the error becomes apparent, the component is repaired. However, at least in the case of top-down debugging, the problem typically requires further analysis and the subcomponents of the faulty component are made the subject of the next level of investigation.

Debugging concurrent programs is more difficult than debugging sequential programs for several reasons, including:

1. The components of a concurrent program are processes, whose input and output are normally dynamic and incremental. This kind of data is more difficult to observe and it is harder to generate in a test context.
2. The processes in a concurrent program often interact in complex ways. To test a process in isolation is seldom practical because its context of use cannot be accurately reproduced.
3. Tracing the execution of a concurrent program is complicated by the fact that it comprises many processes. Unless the traces of the different processes are carefully separated and organised, the trace can be confusing and effective diagnosis very difficult.

It is interesting to consider an analogy with an electrical engineer examining a defective circuit. The engineer has a problem similar to that of the Parlog programmer: to debug a network of concurrently active components. The main ‘debugging tools’ are usually a meter and a set of probes which can be hooked into the circuit’s external and internal ports. Although the engineer does sometimes remove components for testing in isolation, what is interesting is how much of the investigation is done with the components left in place. By attaching probes to judiciously selected ports he can monitor the flow of current and locate the source of most irregularities.

We propose that debugging methodologies for concurrent logic programming should follow suit. As far as possible, processes should be tested in situ, by monitoring dataflow on channels. We call this channel-oriented debugging. Its main advantage is that since testing takes place in situ there is no need to attempt to reproduce the context of use of the process. In particular, it is unnecessary to fabricate input to a component since, unless it is directly affected by the fault, the input provided from its environment should be sufficient. This approach requires that the programmer be given tools analogous to the engineer’s probes and meter: tools which make it possible to probe inside a network, observing the dataflow along internal channels, while still treating each process as a ‘black box’. We describe in Section 5 a channel spypoint facility, which is our attempt to provide such tools for Parlog programmers.

Although we advocate in situ channel-oriented debugging, there will be occasions when it is desirable to test component procedures in isolation. An important special case is the testing of the program at the top level; another application is in bottom-up program development. The key requirement here is to simplify the programmer’s task of managing input and output which is dynamic, rather than static. Our approach is to extend the normal Parlog query system with a set of dynamic input/output facilities, which are described in Section 4.

There remains the problem of execution tracing. With channel-oriented debugging as the first line of attack, execution tracing becomes a secondary, but still important, debugging strategy. We propose some novel software tools to cope with the problem of tracing multiple processes in a selective manner. These are described in Section 6.

In summary, the debugging tools which we propose are as follows:

1. Dynamic query I/O tools simplify ‘black-box’ testing of complete programs or of component procedures in isolation.
2. Channel-oriented debugging tools allow the programmer to probe inside a network, observing the dataflow along internal channels.
3. Process-oriented debugging tools enable the programmer to observe the execution steps of a process.

4. DYNAMIC QUERY INPUT/OUTPUT

Most concurrent logic programming systems provide a query facility similar to that of Prolog: the values of the query’s variables are displayed when the evaluation terminates (assuming that it succeeds). Input, if any is needed, can be typed directly into the query. Thus, a
Prolog programmer might test an append program by entering the query
\[
\text{\texttt{\textless - append([1,2,3],[4,5,6,7],X).}}
\]
in which the test data is represented by the two lists. The programmer hopes that the binding
\[
X = [1,2,3,4,5,6,7]
\]
will be displayed by the system on the termination of the evaluation. However, these 'static' input and output query facilities are not adequate for Parlog, as we explain below.

4.1. Dynamic output
Suppose that the programmer enters the query
\[
\text{\texttt{\textless - sort_randoms(50,SortedInts).}}
\]
as described above. Let us imagine that the computer does not return a sorted list containing 50 integers; instead, the machine just hangs. A static query output facility does not help in this situation. The non-termination means that the programmer will not be able to tell whether the program managed to generate any binding for the SortedInts variable and, if so, what that binding was. Information of this kind can be crucial for effective debugging.

Non-termination is a problem in Prolog programming as well as in Parlog, but there are two major differences. First, a very common reason for the non-termination of a Parlog relation call is suspension; this cannot occur in Prolog. Second, a non-terminating call in a Prolog conjunction blocks out all other calls. In Parlog, other calls may continue to run concurrently with a non-terminating call and its output may indeed affect their execution. Therefore, in Parlog debugging, it is especially important to be able to inspect the output bindings of a call incrementally, before termination.

A simple solution to this problem, which is traditionally used by Parlog programmers, is to add to the query a call to a 'suspendable' output procedure. In this section, we point out some of the deficiencies of this approach and then explain our proposal, which is to build dynamic input/output facilities into the Parlog programming environment.

The most general form of suspendable output procedure is incwrite, which displays a term incrementally as it is produced (i.e. the procedure writes the term from left to right, suspending each time a variable is encountered). By formulating the above query as
\[
\text{\texttt{\textless - sort_randoms(50,SortedInts), incwrite(SortedInts).}}
\]
the incwrite process which runs concurrently with the sort_randoms process will display the terms of the sorted integer list incrementally as they become available. Any data generated by the sort_randoms process will be visible even if it subsequently deadlock.

This approach works, but it has several drawbacks:

1. If the original part of the query (excluding the incwrite call) successfully terminates without producing a ground binding for the displayed variable(s), the incwrite process will remain suspended and cause the query to deadlock.

2. In general, the bindings of many variables should be displayed. Since their size cannot be known in advance, it is preferable to display them in separate windows. Clearly, the window management required to achieve this imposes an undesirable burden on the programmer.

Our approach is to automate the dynamic output facility by incorporating it into the programming environment. The user can request incremental output for all or some variables in a query, and each is displayed in a separate dynamically-created window. The environment takes care of the window management and, when the query terminates, aborts the display and deletes all windows (more accurately, the windows are deleted after the user has had a chance to examine their contents at leisure).

Fig. 8 illustrates this by showing part of a MacParlog screen display. The user has just clicked the query dialogue's Run button after entering the sort_randoms call. Because the dialogue's Incremental view option is selected, MacParlog has automatically created the SortedInts window, into which is being written incrementally the data produced for the variable of the same name.

This incremental view feature offers the user a convenient way to observe 'stream' communication, that is, list data generated by a process which binds the terms of the list successively. Streams are very common in Parlog, so the facility is very useful, but the language supports more general forms of communication. For example, it is quite possible for a process to generate a list containing unbound variables; a trivial illustration of this is the query
\[
\text{\texttt{\textless - X = [99,88,Var1,77,Var2].}}
\]
Unfortunately, our incremental view feature provides little help for such a query, since the display will get no further than the first variable:
\[
[99,88,
\]
Another illustration of the same problem commonly arises if a process incrementally generates some non-list term such as a tree. For example, the query
\[
\text{\texttt{\textless - randoms(50,IntList), make_tree(IntList,Tree).}}
\]
will not be faithfully 'animated' by our incremental view facility, since tree terms are written left to right as for lists, regardless of the chronological order in which
DEBUGGING TOOLS FOR CONCURRENT LOGIC PROGRAMMING

bindings are made to the tree's subterms. Moreover, if the make_tree program has a bug which produces a variable somewhere in the left branch of the tree, the user will never see the right branch. Again, crucial debugging information may be lost.

Our solution to this problem is a second, alternative, form of dynamic display which we call film format. Our query system allows either incremental or film format to be selected for variables in a query. As for incremental view, each filmed variable is displayed in its own special window, which is deleted when the query terminates. The difference is that the window always displays the entire term binding for the variable, including any variables that the term might contain; this is continually rewritten to reflect any further instantiation of variables. The first few showings of the variable Tree during the evaluation of the previous query might be

Tree
  tree(T1,37,empty)
  tree(tree(empty,76,T2),37,empty)
  tree(tree(empty,76,tree(T3,12,empty)),37,empty)

Each version of the term overwrites its predecessor, so that the user literally appears to be watching a 'film' of the variable's changing state as it becomes progressively instantiated by the process.

One disadvantage of both incremental and film format display is the proliferation of windows caused by allocating each variable its own window. For convenience we provide a third format, known as snapshot, which is the same as film format except that many variables can be displayed in a single window, each on a different line, with terms being truncated if necessary.

4.2. Dynamic input

As indicated above, it is possible to provide test input for a Parlog procedure simply by writing the input directly in the query. For example, as a test of the partition procedure, we could run the query

<— partition(5,[7,1,6,4,9],Less,More).

If the machine returns the results

Less = [1,4], More = [7,6,9]

then we will have shown that the procedure is capable of computing at least one correct solution. But such a query would not simulate the normal context of use for the procedure, in which input data is supplied incrementally by another process. For the qsort process shown in Fig. 3, for example, the spawned partition subprocess receives its input list as a stream of integers generated by the randoms process. For maximum concurrency, the partition process should output these terms on the appropriate output stream as soon as they arrive. A defective version of partition might buffer them internally until the input list is completely known; such aberrant behaviour would be unlikely to show up in a test like that above.

More generally, since Parlog processes interact with each other during their execution, the input to a process often depends on its output. Process networks like the one shown in Fig. 9 are quite feasible and even common.

Supplying the input before execution begins is clearly inadequate in such cases.

As with dynamic output, Parlog programmers have found ad hoc solutions to the problem of providing test input to a process incrementally. As we explain next, these are not really satisfactory. We then describe two tools – an in_stream primitive and the concept of film input – which provide much more powerful features.

One way to simulate the normal incremental input is to run a query such as

<— partition(5,[7,1|Rest],Less,More).

Here the list [7,1|Rest] corresponds to the state of the input stream shown in Fig. 10. The first two of possibly many integers for partitioning has arrived on the channel; for a correctly working version of the partition procedure these integers should be output on the respective Less and More channels leaving the process suspended pending the arrival of further data. If the variables Less and More are selected for dynamic output, or if a pair of incwrite cells are programmed into the query, then this behaviour will be fully visible to the user. Unfortunately, the user has no subsequent control over the process, which indeed will deadlock since the variable Rest cannot actually become bound to further data.

What is required is some method for a user to dynamically supply test input for a query variable. We have provided programmers with a simple way to do this by means of a primitive, named in_stream, which allows a stream of terms to be read, incrementally and asynchronously, from the keyboard. In the query

<— in_stream(Integer),
    partition(5,Integers,Less,More).

an in_stream process is run concurrently with the partition process and feeds this process incrementally.

Figure 9. Bidirectional communication between processes.

Figure 10. partition process with incomplete input list.
with its stream input. When the user presses any key a 'pop-up' dialogue appears, as shown in Fig. 11 for MacParlog.

![Figure 11. Pop-up dialogue used by in_stream.]

On clicking Ok any terms T1, ..., Tn typed into the dialogue are added to the Integers stream, by the binding Integers = [T1, ..., Tn | J], and the dialogue disappears. If the variables Less and More have been selected for dynamic output and the partition procedure is functioning correctly then the entered terms should almost immediately appear in the appropriate output windows. The pop-up dialogue can be restored at any time by typing another key, and more terms S1, ..., Sm typed in, resulting in the binding I = [S1, ..., Sm | J], and so on, until eventually the stream is closed by clicking Close, causing the binding J = [ ]. This way, the user retains full control over the input stream and can investigate the dataflow through the partition process throughout its lifetime.

We have found this approach quite effective for program testing and have implemented extensions to in_stream which allow input to be given to any one of a number of streams, and which also monitor back communication from the process to the user.

Although the in_stream solution works well for many examples, several disadvantages are apparent:

1. It can only handle stream input. That is, in_stream(S) always produces a list binding: S = [T[V] when term T is typed, or S = [] when the stream is closed. More general terms such as trees cannot be input incrementally. Moreover, there is no way for the user to subsequently bind any variables in T; only the tail of the stream, V, can be bound later.

2. It cannot easily handle bidirectional communication, in which the user and process need to interact on the same channel. For example, suppose that the user supplies an input binding S = [T[V], whereupon the process makes the binding V = [T[V[1)]; the user must notice this (having selected dynamic output for the channel) and input a dummy term which unifies with T before he can bind V1. Worse, suppose that the process generates a list of terms, of the form [t(U1), t(U2), ...] and expects to receive back communication in the form of bindings U1 = k1, U2 = k2, etc.: the only way for the user to supply these bindings is by typing in the whole list [t(k1), t(k2), ...], which is cumbersome.

Our alternative proposal for dynamic input, which we call film input, seeks to overcome the above problems. Film input allows the user to interact with the query processes via film windows, in a completely general way.

When the query

\[ \leftarrow \text{partition}(5, \text{Integers}, \text{Less}, \text{More}) \]

is entered with dynamic film output requested for all variables, three windows appear, which will display the bindings to variables Integers, Less, and More in film format. However, film windows can give the user write access, as well as read access, to the respective variables.

For example, to enter an initial binding for the Integers variable the user simply selects (using a mouse or cursor keys) the corresponding window and types in the term, say \([7|1]\), directly. This binds Integers to \([7|1]\), and the user can now expect to see the integer 7 output on one of the output streams: a binding \([7|M]\) will be displayed in the More window. A second integer can be supplied on the input stream by again selecting the Integers window, selecting variable 1, and typing in a binding for that variable, e.g. \([1|\text{Rest}]\), and so on.

The Less and More variables can be accessed in the same way, possibly causing the evaluation to fail if the values supplied by the user ultimately clash with those generated by the process.

In general, film input allows the user to select any variable in the current term binding for any query variable, and bind the selected variable to a term of any type. This allows the further instantiation of messages input on a stream, the incremental generation of structures such as trees, and it works equally well for bidirectional communication.

4.3. Summary of dynamic input/output

The film facility for input and output provides a completely general method for dynamic interaction between the user and the query processes. The user can view the entire current term binding of each specified query variable, and can bind any variable in it to any type of term. In this way, the user has the same kind of interaction via each film window as a normal Parlog process has via each of its arguments. Thus, the film/output approach seems to fit very well the analogy of the electrical engineer probing into the wires of a live circuit.

The other forms of dynamic input and output that we have proposed are less general, but they are more convenient in the cases where they are applicable. Incremental output is especially suitable for the display of streams of ground terms; it avoids the unnecessary display of the tail variable of the stream. The in_stream primitive is very convenient for inputting streams of terms that do not need to be further instantiated; this can be done with fewer keystrokes than film input.

5. CHANNEL-ORIENTED DEBUGGING

Viewing the bindings to variables in a query may reveal the presence of a bug. When this happens the programmer should generally begin to scrutinise the network of processes which implements the top-level (query) process. In the case of the sort_randoms program, if the query

\[ \leftarrow \text{sort_randoms}(50, \text{SortedInts}) \]

is entered with dynamic film output requested for all variables, three windows appear, which will display the bindings to variables Integers, Less, and More in film format. However, film windows can give the user write access, as well as read access, to the respective variables.

For example, to enter an initial binding for the Integers variable the user simply selects (using a mouse or cursor keys) the corresponding window and types in the term, say \([7|1]\), directly. This binds Integers to \([7|1]\), and the user can now expect to see the integer 7 output on one of the output streams: a binding \([7|M]\) will be displayed in the More window. A second integer can be supplied on the input stream by again selecting the Integers window, selecting variable 1, and typing in a binding for that variable, e.g. \([1|\text{Rest}]\), and so on.

The Less and More variables can be accessed in the same way, possibly causing the evaluation to fail if the values supplied by the user ultimately clash with those generated by the process.

In general, film input allows the user to select any variable in the current term binding for any query variable, and bind the selected variable to a term of any type. This allows the further instantiation of messages input on a stream, the incremental generation of structures such as trees, and it works equally well for bidirectional communication.

4.3. Summary of dynamic input/output

The film facility for input and output provides a completely general method for dynamic interaction between the user and the query processes. The user can view the entire current term binding of each specified query variable, and can bind any variable in it to any type of term. In this way, the user has the same kind of interaction via each film window as a normal Parlog process has via each of its arguments. Thus, the film/output approach seems to fit very well the analogy of the electrical engineer probing into the wires of a live circuit.

The other forms of dynamic input and output that we have proposed are less general, but they are more convenient in the cases where they are applicable. Incremental output is especially suitable for the display of streams of ground terms; it avoids the unnecessary display of the tail variable of the stream. The in_stream primitive is very convenient for inputting streams of terms that do not need to be further instantiated; this can be done with fewer keystrokes than film input.

5. CHANNEL-ORIENTED DEBUGGING

Viewing the bindings to variables in a query may reveal the presence of a bug. When this happens the programmer should generally begin to scrutinise the network of processes which implements the top-level (query) process. In the case of the sort_randoms program, if the query

\[ \leftarrow \text{sort_randoms}(50, \text{SortedInts}) \]
behaves incorrectly, then the programmer who is conducting a top-down debugging session will switch from the ‘black-box’ view of the program represented by Fig. 7 to the ‘glass-box’ view represented by Fig. 3.

When debugging such a network, it is usually better to focus first on channels rather than on processes. Although it is invariably a process and not a channel which is faulty, channels are conceptually simpler than processes and they are easier to investigate. Channels do not (normally) contain faults but they can provide strong clues about the processes which do. In Fig. 3, we should be able to locate a malfunctioning subprocess by monitoring the dataflow along the IntList channel. If this channel has correct dataflow then the fault lies with the qsort process; if not, there must be an error in the randoms process. There are several alternative ways in which the dataflow might be observed:

1. Run the query
   \[ \langle- \text{randoms}(50, \text{IntList}), \]
   \[ \text{qsort}(\text{IntList}, \text{SortedInts}) \]
   which represents the conjunction of calls to which the query is reduced, and use dynamic output to monitor the variable IntList. Although fine for this example, this ‘macro-expansion’ approach is impractical in general, because the query may be reduced to a conjunction containing a large or unknown number of calls.

2. Run the randoms process in isolation as a query:
   \[ \langle- \text{randoms}(50, \text{IntList}) \]
   and monitor IntList as above. This is quite sound with this example because the randoms process running by itself should behave no differently from when it runs concurrently with the qsort process. In general, however, the ‘isolation’ approach is much more problematic. It can be difficult or impossible to reproduce the original context of a process, because it may interact with other processes in complex and unpredictable ways.

3. Edit the sort_randoms procedure by adding a call to some output procedure such as incwrite, as shown in Fig. 12, so that the IntList channel’s value will be displayed when the sort_randoms query is run. We call this the ‘spy-call’ approach, because the extra call acts like a spy which watches and reports on the channel's dataflow. The spy-call approach has the advantage that the process's context of use is preserved, but it suffers from the same problems as with adding output calls to a query, noted in Section 4.1. An additional drawback is the need to edit and recompile the procedure before and after performing the test.

At run time a window is created when evaluation commits to a clause containing a spied variable, and the variable’s value is displayed dynamically in this window. A dialogue reports when all calls in such a clause body have terminated. The programmer can then inspect at leisure the contents of any windows created for the variables in the clause; when he responds, they are automatically deleted.

In MacParlog, a channel spypoint is set by clicking on the variable’s name in the procedure’s source text. A menu command then summons a dialogue like the one shown in Fig. 13. As the dialogue indicates, the display format for the variable can be any of the incremental, film, or snapshot formats, as described in Section 4.1 for dynamic query output. Using the film format, it is even possible for the user to input data on a channel, as well as to observe it. Clearly, this facility should be used with care, since it could change the behaviour of the program. However, it can be useful in situations such as deadlock: the user can often see the cause of the problem – that some channel has not been bound – and try to break the deadlock by making the intended binding manually.

In general, there is not a one-to-one correspondence between source text variable names and channels, since the same procedure can be called in many different contexts. For example, setting a channel spypoint on the Sorted variable of the qsort procedure is likely to produce a great many windows, each representing the Sorted channel for a spawned qsort process at some level of recursion. However, our implementations provide two ways in which the action of channel spypoints can be made selective, to help reduce screen clutter caused by unwanted windows.

First, the user may only be interested in the channel for the top-level process. This can be achieved by selecting the First context only option in the dialogue, which specifies that the channel spypoint is to be acted upon only in the first chronological context.

Second, the user may specify an arbitrary unification test on the variables of the clause in which the channel spypoint is being set; the spypoint is ignored if the test is not satisfied. For example, suppose we set a spypoint on the variable Sorted in clause 2 of qsort/2, select the All contexts option, and add the condition \( N = 7 \) to the binding of Sorted will now be displayed every time that execution commits to clause 2 of qsort/2 provided that variable \( N \) has the value 7. Fig. 14 shows the screen display at one point during an execution in which two such contexts arise. The dialogue is reporting the
The standard tool for process-oriented debugging in any language is the execution tracer. In a typical Prolog system the tracer displays some or all of the events in the 'life cycle' of each call. Only one call is active at any time in Prolog, whereas in Parlog many calls (processes) may be active concurrently. It is quite possible to represent the Parlog program trace as an interleaving of the traces of individual processes. This is helpful in that the resulting display can provide a useful record of the entire computation in chronological order, but it is generally unsuitable for our purposes since the trace of the single faulty process in which we are interested has to be extracted from a voluminous body of less relevant trace information. In our implementations, the default is that tracing ignores all processes except those for which the user has explicitly set a spypoint. This is a familiar debugging device in other languages, including Prolog. In MacParlog, a spypoint (we call then process spypoints to distinguish them from the channel spypoints described in Section 5) is set by clicking on a relation name in the source text and choosing a menu command.

3. A process spypoint can be made more selective by additionally specifying an optional call pattern; at run time, tracing will be suppressed if the call pattern is not matched.

Figure 15 shows a typical MacParlog screen display during a trace of a qsort process. The process has spawned four subprocesses, and three of these are being traced in their own windows. The trace of the fourth subprocess, for append, will share the window of the parent qsort process. A dialogue has been generated by the tracer which offers the user a set of options for tracing one of the subprocesses.

The main tracing options are:

- **enter**: trace the evaluation of this process, but present a trace dialogue before each subprocess is entered;
- **unleash**: trace the evaluation of this process exhaustively and without offering any further trace dialogues;
- **skip**: do not trace the process but only report its eventual success or failure.

7. FURTHER DEBUGGING FACILITIES

Process-oriented debugging is quite a conventional technique but, by combining it with channel-oriented...
debugging, a further range of useful debugging tools can be developed.

Our process tracer’s most innovative option is call filming, invoked by the trace dialogue’s film button. On selecting this option, a special window is created in which the relation call is ‘filmed’ during the course of its evaluation. ‘Filming’ a call has the same meaning as filming a channel: the current state of the call is repeatedly rewritten into the window, each time overwriting the previous version, so that the call’s changing state is visible in animation.

For the dialogue in Fig. 15, the film button will display a window into which will be successively written the terms:

\[\text{qsort}([7,9,6], [6,7,9])\]

The changing state of the first argument represents the stream of integers arriving from one of the partition process’s output channels. The updating of its second argument reveals a property of the algorithm which the user may not have anticipated: there is no output until the input is completely constructed.

Call filming effectively provides the facility for channel spying within the process debugger. Its availability as a film button in the process tracer solves two problems. First, film offers a useful compromise between the enter option, which may generate too much information, and the skip option, which may generate too little. Second, call filming provides a much clearer view of the dataflow than selecting this option, a special window is created in which the input is completely constructed.

Prolog tracers often provide a ‘print call’ option, which displays the current state of the call being entered. The latter allows the user to give a solution to a call being entered, rather than executing the call, and is convenient for top-down program development: the user can type in the solution for a procedure that is not yet written.

Our film option can be seen as a dynamic version of ‘print call’ which is suitable for Parlog. The ‘unify call’ option too can be generalized to a dynamic form. An emulate option in the process tracer (not illustrated above) would film a call instead of running it (the film option runs the call as well as filming it). Using the call’s film window for dynamic input and output, the user can then mimic the behaviour of the process in a completely general way, finally selecting either a succeed or fail option to terminate the process.

8. CONCLUSIONS

8.1. Summary

We have presented a debugging methodology and a set of debugging tools which we have found useful in our experience of using and teaching Parlog. The methodology relies upon a channel-oriented, in situ, approach to testing as the first line of attack, but with testing in isolation when appropriate, and with possible recourse to process-oriented debugging after the search has been sufficiently narrowed. In support of this approach we propose three kinds of tool:

1. Facilities for observing and providing dynamic input and output on selected variables in a query. The film feature allows completely general input and output of arbitrary terms, while two more specialised facilities are convenient for the input and output of streams: the in_stream primitive and incremental output, respectively.

2. Channel spypoint facilities that allow communication on selected variables to be monitored dynamically.

3. Process tracers which separate the traces of distinct processes into different windows, and which enable the programmer to select only processes and life cycle events that are of interest. A special feature of the process tracer is the ability to film a process: to view communication on its variables while ignoring its internal execution steps.

Almost all of the facilities proposed in this paper have been implemented in two commercially released Parlog language systems: MacParlog (for the Apple Macintosh®) and PC-Parlog (for IBM-PC compatibles). MacParlog and PC-Parlog are available from Parallel Logic Programming Ltd, PO Box 49, Twickenham TW2 5PH, UK. The only exceptions are film input (Section 4.2), which is currently implemented only in PC-Parlog, and the emulate option of the process tracer (Section 7), which has not yet been implemented in either system. These outstanding features are to be implemented in future releases.

Although this paper has concentrated on Parlog, our debugging approach and tools are equally applicable to other members of the family of concurrent logic programming languages (Concurrent Prolog, GHC, Strand, etc.).

Our tools have been designed specifically for use in concurrent logic programming, which lies at the intersection of two distinct disciplines, concurrent programming and logic programming. Nevertheless, in the next two subsections we discuss the wider implications of our work, and some related work, in each of these areas.

8.2. Contributions to concurrent programming

Almost all concurrent programming paradigms allow a computation to be structured into many processes and define at least one method of communication between processes. They differ in the manner in which processes can be created and terminated – for example, whether this is dynamic or static – and in the means of communication.

Clearly, some form of process tracing is applicable to any such concurrent language. A debugger should display the events of each process in a separate window and allow the programmer to focus on the events of interest. In a language such as Occam, where the number of processes is fixed, it is a simple matter to create a window for each process. However, where processes are created dynamically, the number of windows required cannot
be determined in advance. The solution that we adopt is to create for each process a fixed-size window in a convenient position on the screen. The position is computed so as to avoid obscuring other windows as far as possible, but eventually this is inevitable; however, the position, as well as the size, of the trace windows can be changed manually if desired. In a recent microcomputer implementation of CS-Prolog (Kacsuk, pers. comm.), the programmer must specify in advance the number of windows to be used by the process tracer; this method simplifies window placement, but is less flexible.

The main difference between concurrent logic programming and other concurrent programming languages concerns communication. Concurrent logic programming is distinguished by the following characteristic features:

1. Single-assignment variables are used as the communication medium between processes. This means that the value of a variable reflects the entire history of communication on it, which is extremely useful for debugging purposes.

2. The same method of data manipulation is used everywhere in a program: every variable can be considered a channel and, consequently, every procedure a process. This uniformity minimises the number of separate debugging concepts that are required.

Some other concurrent languages exhibit one of these properties, but not both. For example, PCN\(^4\) has adopted single-assignment variables for inter-process communication, but allows destructive assignment to ‘mutable’ variables within a process, thereby differing on the second point. The second advantage would be shared by, for example, a subset of Ada in which inter-process communication is done solely via global mutable variables.

Many concurrent programming languages are based on the message passing paradigm, in which processes communicate by explicitly sending and receiving messages. Examples include even some logic programming languages such as Delta Prolog\(^8\) and CS-Prolog\(^9\) which, despite having a concurrent behaviour, differ fundamentally from languages like Parlog. In these, each process sequentially executes a Prolog computation and may exchange messages with other processes by calling special non-logical primitives. Our channel-oriented debugging tools could easily be adapted for use in any message passing language that uses channels, to monitor the messages passed along each channel. In the CS-Prolog debugger, a similar effect can actually be achieved within the process tracer, by ‘spying’ the message passing primitive, thus reporting each time a communication takes place.

8.3. Contributions to logic programming

The approach to debugging that we have considered is primarily procedural. Procedural methods remain the most common approach to debugging logic programs. In the case of Prolog, there is the ubiquitous ‘four port’ debugger\(^2\) from which our Parlog process tracer is derived. Perhaps the most advanced procedural debugger for Prolog is the Transparent Prolog Machine,\(^9\) which displays a graphical representation of the computation.

Some Prolog systems, e.g. Sicstus Prolog, NU-Prolog, etc. include coroutining control facilities which allow the programmer to force goals to delay until their arguments are sufficiently instantiated. Programs that make use of coroutining can be very difficult to debug using the conventional Prolog debugger because the trace of a woken goal appears in the midst of the trace of the procedure which made the binding that woke it. This problem is quite similar to that of debugging concurrent logic programs with a single-window tracer, so an obvious development would be to adapt our multi-window process tracer for use with coroutining languages: a new window would be opened whenever a goal suspends, which can be used to trace the goal when it is woken.

An idea similar to our channel spypoints has been suggested for Prolog as an aid to debugging non-terminating programs.\(^2\) A procedure argument can be spied, to display its binding at regular intervals, to check on the behaviour of the procedure. The main difficulty in Prolog is the fact that variables lack the single-assignment property: their bindings can be undone on backtracking. The same paper also suggests the monitoring of Prolog’s I/O channels in separate windows, a concept that is even closer to our channel spying.

One alternative to procedural debugging is static analysis, which can reveal some kinds of error at compile time, perhaps by checking types or modes of procedures. A logic programming system that includes static, as well as procedural, debugging tools is the NU-Prolog Debugging Environment (NUDE).\(^{15}\)

The fact that logic programming languages have a declarative, as well as a procedural, semantics is exploited by a third approach known as declarative (or algorithmic) debugging.\(^{16}\) This attempts to locate errors in a program by comparing the actual result of a call with the result expected by the user. Huntbach\(^13\) has extended Shapiro’s approach to cover concurrent logic programming languages.

8.4. Closely related work

The work closest to ours is a debugger for GHC which is currently being developed (Trehan, pers. comm.). This is intended to display a graphical representation of the evolving process network, showing both processes and channels. The user can select a process to execute, or select a channel to view its current binding. The debugger displays similar information to ours, except in a graphical form, but it lacks a facility for dynamically displaying variable bindings. It has the novel ability to display the part of a variable binding that has already been ‘consumed’ (matched against) by some process; however, this concept is hard to define and is implemented only for the special case of lists.

In another recent development, Brayshaw\(^1\) describes a system for visualising Parlog program execution. Its main feature is a graphical display of the computation, in which each node shows the name and status of the corresponding goal. The programmer can zoom in on selected nodes to obtain more fine-grained information about the goal’s current status. The system provides a choice between two representations, both built from the same kinds of node: (1) an AND/OR tree, and (2) a process network, where the nodes are linked by channels
labelled by their current bindings. The latter is quite similar to Trehan's GHC debugger.

Both of the above systems are (at least currently) based on the idea of 'replaying' a pre-recorded execution history; programmers cannot interact with a live program. This method makes impossible many of the capabilities which our tools provide (e.g. film input).

8.5. Future work

There is ample scope for improvements to the debugging tools that we have developed. One weakness is the display of unbound variables as their runtime addresses; the address of a variable may change when it becomes shared with another variable or when garbage collection occurs. This can be misleading, especially in the process tracer, since different addresses (printed at different times) may actually denote the same variable. It would be better to display each variable by an unchanging identifier, e.g. the identifier used for channel spying. Another weakness is that our implementation of channel spypoints delays the display of a spied variable until (if ever) the clause is selected for commitment. This is a problem only when the spied variable occurs in a clause guard (an uncommon case), but its solution merits further research.

We have emphasised the benefits for debugging of single-assignment variables as a communication medium: the entire history of communication on a channel is retained and displayed in its dynamic display window. In some circumstances the sheer size of the history could overwhelm the programmer, particularly 'film' display of a very large term containing a few unbound variables, especially if they are near the beginning of the display. It would be beneficial to provide additional tools to concentrate a film display on the most interesting parts of a term, i.e. the variables in it, without losing their context.

Our existing tools represent clearly the process interpretation of Parlog programs: one window shows the execution of a process or the communication on a channel. But there is another interpretation — as a proof — which is not currently represented: it would be useful to display the proof tree corresponding to the execution of a program. Ideally, the tracer should integrate the process view with the proof tree view and allow the user to switch between them. Brayshaw's system, mentioned above, appears to fulfil this need, but in a different framework from ours.

Finally, it would be worth investigating alternative approaches to debugging concurrent logic programs. Procedural methods have limitations, especially in concurrent programming, where it is possible and even normal for the same program to behave completely differently on different runs, and therefore difficult to reproduce bugs for investigation by procedural debugging tools. One solution is to check all of the possible non-deterministic behaviours of a program, an approach considered by Gregory et al.,

However, this would be tractable only for testing small procedures in isolation, not for an entire large program. More ambitiously, it may be possible to develop a form of declarative debugging for concurrent programs, in which the program's actual behaviour is checked against a specification of its behaviour, including the temporal properties, supplied by the programmer.

REFERENCES