COD: a channel-oriented debugger for concurrent logic programs

Steve Gregory

Department of Computer Science
University of Bristol
Bristol BS8 1TR, England

steve@cs.bris.ac.uk
http://star.cs.bris.ac.uk

We present a debugger for concurrent logic programs that works on the *channel-oriented* principle: it allows program bugs to be diagnosed by probing values of variables, rather than by the more traditional method of tracing the execution of processes. A key feature of our debugger is that, like Shapiro's algorithmic debugging technique, it searches for a fault in an "intelligent" manner, asking the user pertinent questions about the program's correctness. Whereas algorithmic debugging searches for a faulty node in a proof tree, our debugger searches for a faulty node in the program's process graph. We claim that this allows an observed bug to be rapidly traced back to the fault of which it is a symptom. A prototype implementation has been developed for the KLIC dialect of KL1.

1 Introduction

1.1 Process-oriented debugging

Most debugging tools for logic programs are *process-oriented*: they allow the user to trace the execution of a program while observing enough information about the execution to locate a bug that has been observed. For example, Prolog's four-port debugger provides facilities for the user to traverse the proof tree and the search tree defined by the program, in the usual top-down left-right order of Prolog.

Prolog's debugger essentially allows the user to manually search the proof tree for a *buggy* node: for example, one that produces a runtime error, unexpectedly fails, or succeeds with incorrect output bindings. A typical search strategy is as follows:

1. Select a subtree to search: either the entire tree or a subtree rooted at a spypoint, preferably near the buggy node.
2. Starting from the root, **skip** each of the offspring nodes until a buggy node is found.
3. Use the **retry** option to enter the subtree rooted at the buggy node.
4. Repeat from step 2 until a buggy node is found that has no buggy offsprings.

Many concurrent logic programming systems, including PC-Parlog (PLP, 1989), Parallel Parlog (Crammond *et al*., 1989), and KLIC (Chikayama, 1994), include debugging tools that work in a similar way, but which are tailored to the different operational semantics of the language. The basic debugging strategy, manually traversing a proof tree to locate a buggy node, is the same as in Prolog. However, experience suggests that this technique is less suitable for concurrent logic programs than for Prolog programs, for the following reasons:

1. The **retry** command, which is very useful in debugging Prolog programs, cannot be implemented directly because there is no built-in backtracking facility. If a **skip**ped goal is found to be buggy, the user has to restart the computation and **skip** to the same point again before **enter**ing the buggy node.
2. In concurrent logic programs, a **skip**ped goal does not necessarily terminate before the next one is started. The user may need to **skip** several goals before any of them exhibits an error, and it can be hard to keep track of the current status.
3. Many debuggers for concurrent logic programs do not report the termination of each goal, because recording this information can be very expensive in memory consumption. This makes it even harder to keep track of a **skip**ped goal.
4. A Prolog debugger allows the user to step through a single execution sequence but, for concurrent logic programs, the user may need to interact with many goals simultaneously. A multi-window display facility is virtually essential for this; interleaving concurrent trace output would be hopelessly confusing. Moreover, a window created to trace a goal must be deleted when it terminates, which requires the feature of (3) above.

A fundamental feature of process-oriented debugging is that a buggy node located by tracing is not necessarily the one that is the real cause of the bug. A goal often behaves wrongly simply because its input arguments are incorrect, and these may be produced in a completely different part of the proof tree.

For example, consider a compiler[1] defined by the following clause:

```
compiler(Source, Object) :-                                    (*)
    canonicalize(Source, CanonicalForm),
    generate(CanonicalForm, NaiveObject),
    analyze(CanonicalForm, OptInfo),
    optimize(NaiveObject, OptInfo, Object).
```

If the output of the compilation, `Object`, is incorrect, the user will most probably **enter** the proof tree subtree representing the goal `optimize(NaiveObject, OptInfo, Object)`, and eventually find the buggy node that incorrectly bound `Object`. However, this incorrect binding may well have been caused by an incorrect value for `OptInfo`, which was produced by `analyze(CanonicalForm, OptInfo)`: the *cause* of the bug was in `analyze/2`, but it was only noticed in `optimize/3`.

In general, after finding a node that is “buggy” only because of its input values, the user has to start again to search for the proof tree node that produced the incorrect bindings; this may need to be repeated many times until the cause of the bug is located. We shall refer to the proof tree node that is ultimately responsible for the bug as the *faulty* node.

1.2 Algorithmic debugging

Algorithmic, or *declarative*, debugging was originally proposed by Shapiro (1983) and has also been adapted to concurrent logic programming languages (Huntbach, 1987; Takeuchi, 1992).

The overall aim of algorithmic debugging is the same as that of the process-oriented debugging approach described above: to locate a faulty node in the proof tree. However, it does so in a completely different way, ignoring the operational semantics and focusing on the

[1] This example was suggested by Takashi Chikayama (personal communication).

declarative reading of a program. By asking the user a succession of questions about the correctness of the substitution at each proof tree node, it can automatically zoom in on a faulty node. Roughly, a node (representing a clause instance) is considered faulty if its substitution is incorrect but those of its offspring nodes are all correct. In the compiler example (*), an algorithmic debugger would discover that the output of the `analyze(CanonicalForm, OptInfo)` goal is incorrect and thereby deduce that the faulty node lies in the subtree rooted at that goal. By checking the output substitution of each subgoal of `analyze(CanonicalForm, OptInfo)`, the exact faulty node can be located.

The main differences between algorithmic debugging and conventional process-oriented debugging are:

- Algorithmic debugging is independent of the language's operational semantics. This avoids some of the problems enumerated in Section 1.1: for example, no interactive multi-window display is needed.
- Algorithmic debugging directly locates a faulty node without the need to search for it via a chain of (symptomatic) buggy nodes. In some ways, this is an advantage. On the other hand, since the search for a faulty node is completely unguided by knowledge of any observed bug, the user may need to answer many, seemingly irrelevant, questions. The faulty node that is found may even be unrelated to the observed bug; this can be counterintuitive.

Perhaps the most serious drawback of algorithmic debugging is that it relies on the "oracle" (usually the user) to verify the correctness of the substitution generated at each node. Although this is possible in principle, the arguments of the goal may be so large or complex that it is impractical. If the user fails to identify the incorrectness of a substitution, declarative debugging will not be able to locate the bug. With the manual approach to debugging, if a bug is accidentally missed, at least the user can obtain some information about it before starting again.

1.3 Channel-oriented debugging

In (Conlon and Gregory, 1992) two kinds of debugging tools for concurrent logic programs were discussed: "process-oriented" and "channel-oriented" debugging. Whereas process-oriented debugging is based on the traversal of a proof tree, channel-oriented debugging views a program as a graph, in which nodes represent goals and arcs represent shared variables. It was argued that channel-oriented debugging can be a much more natural way to debug concurrent logic programs than the process-oriented method.

The basic idea behind channel-oriented debugging is to search for a bug by observing values of variables. If the output of a goal is incorrect, then either one or more of the inputs are incorrect, in which case the producers of those incorrect values must be investigated, or the inputs are correct, in which case the fault lies in this goal. The user can "look inside" the clause body (subgraph) to which a faulty goal expands, and then locate the faulty subgoal by examining the variables in that clause body, and so on.

PC-Parlog and MacParlog (PLP, 1989) allows the user to set "channel spypoints" on any variables in a program before it is run. When such a variable is created at runtime (i.e., when the clause to which it belongs is invoked) its value is displayed in its own window, and is frequently updated as the variable becomes further instantiated. One problem with this is that the user has to decide (before running the program) which variables to spy. PIMOS (Chikayama, 1992) provides a more interactive facility whereby the user can select the variables to view during process tracing. All existing facilities for channel-oriented debugging suffer from the drawback that the user has to decide which variables to view in order to diagnose a bug.

2 Systematic channel-oriented debugging

We propose a debugger that will provide a systematic way to perform channel-oriented debugging:

- Like the channel-oriented debugging tools described in Section 1.3, our debugger aims to locate a faulty node in the *process graph* representing the program.
- Like algorithmic debugging (Section 1.2), our debugger attempts to locate an error *automatically*, by asking the user pertinent questions about correctness.

2.1 What is a bug?

We define a bug to be any kind of unexpected behaviour observed by the user while running a program. In a concurrent logic program, this could include any of the following:

- **Failure**. Failure is a fatal error in a flat concurrent logic programming language. For a goal to fail, at least one of its arguments must have an incorrect value.
- **Perpetual suspension**. KLIC (Chikayama, 1994) detects perpetually suspended goals: goals that are suspended on variables which will never be instantiated. Each such goal has at least one argument that is incorrectly uninstantiated, i.e., unbound when it should be bound.
- **Deadlock**. Deadlock is a special case of perpetual suspension in which *all* goals are perpetually suspended. If deadlock occurs, each of the deadlocked goals has at least one argument that is incorrectly unbound.
- **Run-time error**. Other kinds of run-time error (e.g., “file not found”) may be caused by
  (a) an incorrect value for a goal’s argument (e.g., the filename), or
  (b) a problem with the environment (e.g., the file has been accidentally deleted).
- **Incorrect answer**. Interactive concurrent logic programming systems often display the answer substitution for variables in a query. If this answer is incorrect, obviously one or more of the query variables has an incorrect value.
- **Incorrect output**. Input and output in concurrent logic programs should ideally be performed via a single long-lived goal, say `io`, which takes a stream of I/O requests as its argument. KLIC encourages this approach by providing *only* this form of I/O primitive. If a program produces incorrect output, this must be because the argument of the `io` goal has an incorrect value.

This analysis shows that almost every bug is characterized by an incorrect binding, or a lack of a binding, for a variable. The only exception is the “environmental” form of run-time error of 4(b) above. (We will assume in this paper that our language is flat and only allows I/O to be done via streams.) Our debugger tracks down the cause of an incorrect variable binding by traversing the process graph, as described below.

2.2 Searching the process graph

Assume for now that the program’s process graph is static and directed. Debugging begins by marking the arc representing the incorrectly bound variable. Now the system identifies the producer of this variable: the node from which the marked arc is output. The producer goal, with its arguments, is presented to the user, who decides whether the goal is correct:

- If the input arguments are correct, the selected node is identified as the faulty node.

- If one or more of the goal's input arguments is incorrect, the mark is moved to an incorrect input argument, and the search continues in a similar way, asking the user about the correctness of the goals that produced those arguments. This continues until a faulty node — one with no incorrect inputs — is located.

In general, process graphs are not static because concurrent logic programming languages allow processes (goals) to be spawned dynamically. We handle this by viewing the graph as 3-dimensional: At the lowest level, each node is a built-in goal or an unexecuted user-defined goal. A subgraph that corresponds to the body of a clause can be abstracted by a single node corresponding to the clause head. For example, given the program of Figure 1, Figure 2 shows three alternative views of the process graph for the goal `partition([8],11,L,X)`.

```
partition([X|L],Y,XL1,L2) :- X=<Y | XL1 = [X|L1], partition(L,Y,L1,L2).
partition([X|L],Y,L1,XL2) :- X>=Y | XL2 = [X|L2], partition(L,Y,L1,L2).
partition([],_,L1,L2) :- true | L1 = [], L2 = [].
```

Figure 1: The `partition/4` procedure

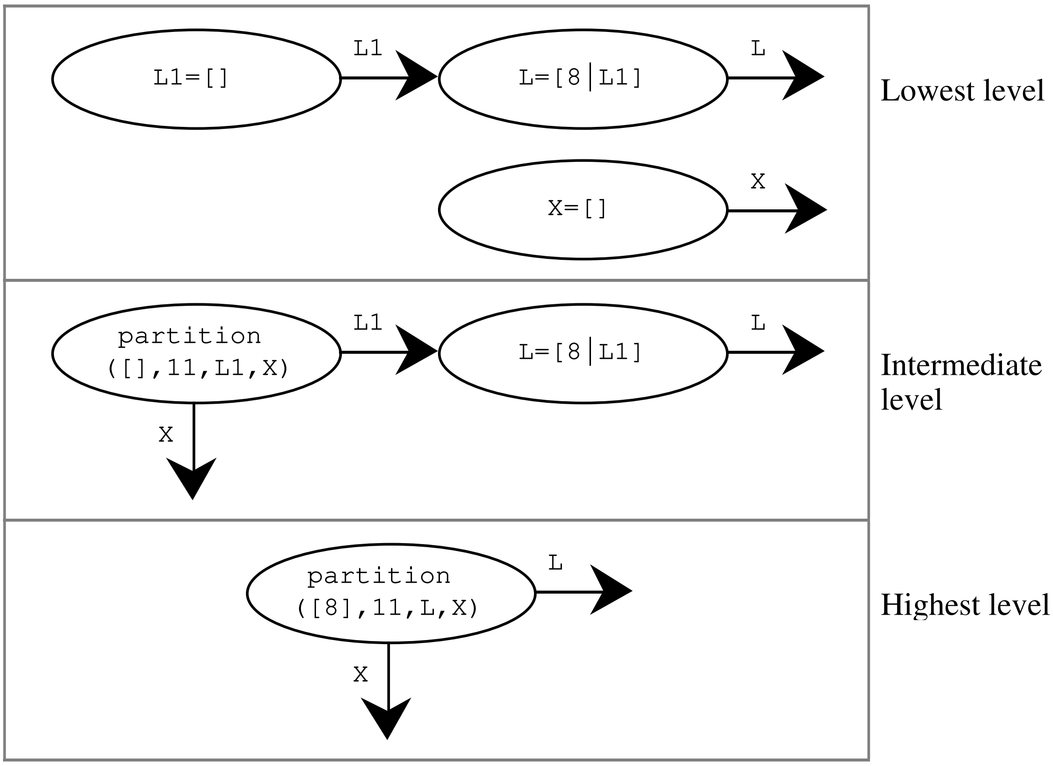

Figure 2: Process graphs for the goal `partition([8],11,L1,X)`

These alternative views mean that, in general, a variable `X` can have many producer goals: from the primitive goal that binds the variable (often '='), through its ancestors, to the goal that appears in the clause body where `X` was created. In the above example, three goals all act as producer of `X`:

`X=[]` `partition([],11,L1,X)` `partition([8],11,L,X)`

If `X` has an incorrect binding, our debugger presents *each* of these goals to the user to check whether any of them has incorrect input arguments.

To illustrate the use of our debugger, we introduce a bug into the `partition/4` procedure by adding an extra, third, clause, as in Figure 3.

```
main :- list30(L), qsort(L,S), io([print(S),nl]).

list30(L) :- true |
   L = [27,74,17,33,94,18,46,83,99,2,32,11,28,85,99,47,28,82,6,53,11,
        28,61,74,18,92,40,53,59,8].

qsort(L, R) :- true | qsort(L, R, []).
qsort([X|L],R,R0) :- true |
   partition(L, X, L1, L2), qsort(L2, R1, R0), qsort(L1, R, [X|R1]).
qsort([],R,R0) :- true | R = R0.

partition([X|L],Y,XL1,L2) :- X<Y | XL1 = [X|L1], partition(L,Y,L1,L2).
partition([X|L],Y,L1,XL2) :- X>Y | XL2 = [X|L2], partition(L,Y,L1,L2).
partition([X|L],Y,L1,L2) :- X=:=Y | partition(L,Y,L1,L2).
partition([],_,L1,L2) :- true | L1 = [], L2 = [].
```

Figure 3: A buggy quicksort program

The `main/0` procedure in Figure 3 will print a sorted copy of a given list of numbers that erroneously has duplicates removed so that, instead of the correct value

```
[2,6,8,11,11,17,18,18,27,28,28,28,32,33,40,46,47,53,53,59,61,74,74,82,
 83,85,92,94,99,99]
```

the program prints the list

```
[2,6,8,11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99].
```

Figure 4 shows the debugger in action: the single-character commands here are input by the user; everything else (except the comments) is output by the system. First the argument of the `io` goal is displayed: this is a term containing the sorted list. The user notices that the fifth member of the sorted list is wrong (it is 17 instead of 11), so the sublist beginning with 17 is selected in the next few steps. The user then asks to see the producer of the incorrect list `L`, and the system displays the goal `L = L`. This is not very informative, so the parent goal, `qsort([],L,L)`, also a producer of `L`, is selected by the '`u`' ("up") command. This goal is correct, but one of the input arguments is wrong. The user asks to see the producer of the first argument, an empty list, and the system displays the trivial goal `[] = []`. The ancestors of this goal, which are also producers of the offending empty list, are displayed in turn by the '`u`' command. Finally, the goal `partition([11,8],11,[8],[])` is found, which is clearly incorrect, signalling that the problem lies in the `partition/4` procedure. It is then a simple matter to discover that the third `partition/4` clause is responsible for the problem.

In our compiler example (*), the user would notice that the final output variable, `Object`, has an incorrect value, and select the subterm of `Object` that is incorrect. The system would then locate the subgoal of `optimize(NaiveObject, OptInfo, Object)` that made the incorrect binding. That subgoal behaves incorrectly because of an incorrect value inside the `OptInfo` term: the system would then jump directly to the subgoal of `analyze(CanonicalForm, OptInfo)` which incorrectly bound `OptInfo`, and rapidly locate the cause of the problem.

```
Current term is                % This is the io argument
[print([2,6,8,11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,
        99]),nl]
1
1th argument is
print([2,6,8,11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,
       99])
1
1th argument is                % The argument of print is the sorted list
[2,6,8,11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99]
2
2th argument is                % Move along the list to the first incorrect item
[6,8,11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99]
2
2th argument is
[8,11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99]
2
2th argument is
[11,17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99]
2
2th argument is                % This list is wrong: it begins with 17
[17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99]
g
Producer is                    % See who produced this list
=([17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99],
  [17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99])
u
Parent goal is                 % Look at parent goal (in proof tree)
qsort([],[17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99],
         [17,18,27,28,32,33,40,46,47,53,59,61,74,82,83,85,92,94,99])
1
1th argument is []             % qsort goal is correct but 1st argument is wrong
g
Producer is =([],[])           % See who produced this empty list
u
Parent goal is partition([],11,[],[])       % Look at parent goal
u
Parent goal is partition([8],11,[8],[])     % Look at parent goal
u
Parent goal is partition([11,8],11,[8],[])  % Look at parent goal
                                            % This is wrong!!!
```

Figure 4: Debugging the quicksort program

2.3 Modes

For the sake of simplicity, we have assumed above that the process graph is directed: i.e., each variable has a known producer goal. This is reasonable when all variables are instantiated: whenever a variable is bound, the system records which goal bound it. If the variable's value proves to be incorrect, the recorded goal will be treated as the producer and investigated. However, in order to track down the cause of problems such as deadlock, our debugger also needs to locate bugs that cause a variable to be erroneously *uninstantiated*.

Many concurrent logic programs enjoy the property that a compile-time analysis can determine a single producer goal for each variable. If this is so, unbound variables can be treated in the same way as bound variables: if a variable is unbound when it should be bound, the goal that is responsible for binding it is known to be faulty and is investigated.

Unfortunately, it is not possible to rely on programs having the single-producer property: even if programs were restricted to satisfy it, a *buggy* program may not. In the worst case, *all* goals that share an uninstantiated variable must be considered potential producers of its value: the user must investigate each of the potential producers in turn to discover the reason for a variable being erroneously unbound. Ideally, in order to minimize the search, the system should eliminate as many of the potential producers as possible.

3 Method of use

Our debugger executes a program while recording information about producers of variables, as described in more detail in the next section. In principle, this information could be collected during *every* execution of the program. However, as this consumes memory space, it is more practical to invoke the debugger only when it is needed. The method of use that we envisage is that:

1. a program is executed until unexpected behaviour is observed;
2. the buggy execution is repeated with the debugger active;
3. the variable that has an incorrect binding, or is incorrectly unbound, is selected and then investigated by the user.

As with most debugging tools, it is necessary to repeat an execution in such a way that the incorrect behaviour is reproduced. The problem is that concurrent logic programs are non-deterministic, especially when executed on parallel architectures, so bugs can be hard to reproduce. However, we have previously developed an Instant Replay tool (Shen and Gregory, 1995) that cheaply records a program's execution so that it can be replayed for debugging purposes. We assume that the buggy execution observed in (1) above has been recorded in this way.

When a bug is observed, the user must select the variable that has an incorrect value so that its cause can be investigated. The way to do this depends on the type of bug, of those identified in Section 2.1:

- **Incorrect answer**. An incorrect answer substitution can only be seen after the query has successfully terminated: the incorrect variable can then be selected and the query reexecuted.
- **Failing, perpetually suspending, deadlocked, or error goal**. We assume that, when such an error occurs, the language system will invoke a process that allows the user to select a variable from the offending goal's arguments and begin debugging. Systems such as KLIC already perform a limited amount of error handling in these cases, even if this is only to display the offending goal.
- **Incorrect output observed after successful termination**. We assume that the `io` goal, which performs the output, will continue running after normal termination of the program. If incorrect output was observed, the user can select the incorrect variable from the `io` goal's arguments.
- **Incorrect output observed during execution**. Incorrect output may be noticed during a long or indefinite execution. In this case the user should interrupt the program. We assume that the `io` goal will continue running after interruption, so that debugging can begin if required.

4 Implementation

A prototype implementation of our debugger has been developed for KLIC, which is an ideal host system because

- it is a flat language;
- input/output is done via a stream-oriented `io` predicate (actually named `klicio`);
- it has a `postmortem` facility which allows any goal to be executed after normal or abnormal termination — this can be used to handle error goals and to execute the `io` goal after interruption, as explained in Section 3.

Our debugger begins by performing a source-to-source translation of the KLIC program to be debugged: a preprocessor analyses the program's clauses to determine the producer occurrence of every variable, and labels each variable with a term that represents its producer goal. Clauses are also modified so that these labels are ignored by matching and primitive goals. The translated program is compiled and linked with the interactive debugger to produce an executable program which, when executed, allows the user to check for errors in the argument of the `io` goal.

4.1 Modes

For our prototype implementation we assume that the I/O mode of each goal is fixed and can be determined at compile time. As explained in Section 2.3, this assumption is not generally true, but it simplifies the implementation: most of the work can be performed at compile time by the preprocessor.

As a further simplification to the prototype implementation, we require the I/O modes to be declared by the programmer. This is not a safe approach because it is possible for the programmer to make mistakes with the mode declarations as well as (or instead of) the bugs in the program itself. A better method would be to perform an *automatic* mode analysis of the program wherever possible; we return to this point in Section 5.2.

4.2 Naming goals

During a debugging execution, each variable is labelled with a term that represents its producer goal(s), so that the debugger can jump from the variable to its producer. We have experimented with two different representations for goals. In the first method, a goal `p(X,Y,Z)` is represented simply by the term `p(X,Y,Z)`. Unfortunately, this method has the disadvantage that potentially the whole proof tree has to be stored in memory, including all variables ever used throughout the computation, because these cannot be garbage collected. For example, for the goal `p(X,Y,Z)`, as long as `Z` is referenced, so also are `X` and `Y`.

To reduce the memory overhead, a variable can be labelled by a unique identifier for a producer goal, instead of the goal itself. This identifier allows the goal to be found later if required. The only memory overhead is then the storage of goal identifiers; garbage collection is not affected. The disadvantage is that it may take longer to jump from a variable to its producer: the system must automatically restart the computation and locate the producer goal from its identifier (using our Instant Replay tool to guarantee deterministic execution).

One unique identifier of a goal is its "pathname" in the proof tree. For example, the pathname of the top-level goal `main` is `[]`; the pathname of the second subgoal of `main` is `[2]`; the first subgoal of this has pathname `[1,2]`; and so on. The worst-case memory requirement for storing the pathnames of *all* goals is proportional to the size of the proof tree (the number of reductions); the memory occupied by each individual pathname is proportional to its depth in the tree, which has a similar worst case.

Both of these schemes have been implemented. Figure 1 compares the memory requirements for the original program and for the preprocessed program using each of the two recording methods. The example used was the bug-free quicksort benchmark, to sort lists of length ranging from 100 to 500. Each row of the table shows a different problem size, given in the first column. The second column shows the heap size (to the nearest 100 words) required by the original quicksort program; the third and fourth columns show the heap size (to the nearest 1000 words) required for the versions which annotate variables with producer goals and pathnames, respectively.

size	original	goals	pathnames
100	300	35000	5000
200	500	82000	8000
300	700	143000	12000
400	900	215000	16000
500	1100	300000	20000

Figure 5: Comparison of recording schemes for channel-oriented debugger

As expected, storing pathnames rather than goals results in much better memory performance. Although the storage of pathnames incurs an overhead, the memory used increases with the size of the problem at the same rate as in the original program, whereas the memory required for the naive scheme increases much more rapidly.

5 Conclusions

5.1 Current status

As outlined above, a prototype implementation of our debugger has been developed for KLIC. This comprises a preprocessor, which translates a KLIC program to an annotated version, and a run-time system, which is linked with the annotated program. The run-time system defines the procedures introduced by the preprocessor and, in particular, provides a special version of the `klicio` primitive which allows the user to identify and diagnose any errors in its argument. The only kind of bug that the current prototype can handle is incorrect output produced (by `klicio`) after successful termination.

Our prototype has served to demonstrate the value of the channel-oriented debugging scheme and to test its feasibility. If the pathname method is used to identify producer goals, we believe that the system should be practical for quite large programs.

5.2 Future work

There are several ways in which our current implementation could be improved:

- To allow the diagnosis of incorrect output produced during execution, the execution of the `klicio` goal needs to be continued after interruption by the user. This can be done by using the `postmortem` facility of KLIC.
- To allow failing, perpetually suspending, and error goals to be diagnosed, we need to modify KLIC's error handler so that the offending goal's arguments are displayed for debugging purposes. KLIC's `postmortem` facility should allow this to be done.

- Our current prototype only works for programs that are well-moded, in the sense of (Ueda and Morita, 1994). However, our system should also handle programs that are not well-moded: even if the language requires programs to be well-moded, the program to be analysed may not be because it presumably contains bugs. More complex run-time support is required to handle non-well-moded programs: the producer goal for each variable must be determined and stored dynamically.
- In our current prototype, I/O modes must be declared by the programmer. However, these modes should ideally be inferred automatically. For well-moded programs, we should be able to adapt Ueda and Morita's mode analysis algorithm for Moded FGHC, which is already being modified for KL1: this algorithm seems to be able to find the information that we require about the producer of each variable.
- Ueda and Morita's mode analysis algorithm only works for well-moded programs. However, it is desirable to extract as much mode information as possible even from non-well-moded programs, as suggested in Section 2.3. This may require many changes to Ueda and Morita's algorithm.
- The method for naming producer goals by pathnames (Section 4.2) could probably be further optimized: for example, a pathname could extend its parent's by just *log N* bits, where *N* is the number of goals in the clause body.

5.3 Related work

Our channel-oriented debugging scheme is, in some respects, a reinvention of Pereira's (1986) "rational debugging" for Prolog programs: both systems record dependencies between variables and their producers so that the cause of incorrect variable bindings can be determined. One difference is that our system is intended for concurrent logic programs rather than Prolog: this means that the view of a computation is somewhat different, as is the debugger's method of use. Our implementation is also very different from Pereira's debugger, which employs a modified unification algorithm that records term dependencies entirely dynamically.

Acknowledgements

I am grateful to Takashi Chikayama for his comments on the subject of this paper, and to Kish Shen for general discussions on debugging concurrent logic programs.

This work has been supported by ICOT as part of the Fifth Generation Computer Systems Follow-on Project.

References

T. Chikayama. 1992. Operating system PIMOS and kernel language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, June 1992), H. Tanaka (Ed.). Tokyo: ICOT, pp. 73–88.

T. Chikayama. 1994. *KLIC User's Manual*. Tokyo: ICOT.

T. Conlon and S. Gregory. 1992. Debugging Tools for Concurrent Logic Programming. *Computer Journal 35*, 2, pp. 157–169.

J. Crammond *et al*. 1989. *The Parallel Parlog User Manual*. Department of Computing, Imperial College, London.

M.M. Huntbach. 1987. Algorithmic Parlog debugging. In *Proceedings of the Symposium on Logic Programming* (San Francisco, August), S. Haridi (Ed.). IEEE Computer Society Press, pp. 288–297.

L.M. Pereira. 1986. Rational debugging in logic programming. In *Proceedings of the 3rd International Logic Programming Conference* (London, July), E.Y. Shapiro (Ed.). New York: Springer-Verlag, pp. 203–210.

PLP. 1989. *Hands on PC-Parlog*. Twickenham, UK: Parallel Logic Programming Ltd.

E. Shapiro. 1983. *Algorithmic Program Debugging*. Cambridge, Mass.: MIT Press.

K. Shen and S. Gregory. 1995. Instant Replay debugging of concurrent logic programs. *New Generation Computing 13*, 4, to appear.

A. Takeuchi. 1992. *Parallel Logic Programming*. John Wiley and Sons.

K. Ueda and M. Morita. 1994. Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing 13*, 1, pp. 3–43.