Implementing Zero Overhead Exception Handling

S. Drew, K. J. Gough, J. Ledermann*

SUMMARY

In programming language design, an increasing emphasis is being placed on providing facilities for improving program robustness and reliability. To this end many languages provide mechanisms for handling exceptional events which arise during program execution. The design of exception handling models have been widely discussed since the mid nineteen-seventies. Many authors have argued that the introduction of exception handling into a program should have as little effect as possible on the performance of programs which do not raise exceptions.

In this paper we review the known theory of low-overhead exception handling, and give implementation experience of a zero overhead exception handling system. Finally we review the subtle influence which such exception handling has on other phases of code generation.

Keywords: exceptions, exception handling, programming languages.

INTRODUCTION

A number of contemporary programming languages provide for exception handling, either as a facility in the base language[1], or as an extension to a previously existing base language[2]. The rationale for placing such facilities in these languages is invariably based on the argument that such facilities provide the means to construct robust and reliable programs[3].

In this context, by an **exception** we mean an abnormal event which is detected during the execution of a program, and which prevents the progam from continuing along its normal execution path[4, 1]. We do not distinguish between such events which are detected by the hardware, or by software tests automatically inserted by the compiler, or by explicit tests which are encoded in the source of the program. An execution environment which allows a program to regain control of the execution following the detection of an exception is said to provide **exception handling**.

Different exception handling models have provided a wide variety of ways of associating exception handlers with various regions of the program, and provide a diverse repertoire of possible response types. Elsewhere we have suggested a taxonomy of response types[5].

Here however, our concern is with implementation mechanisms, and we assume a simple execution model which is able to be extended to other cases in obvious ways. Firstly, we are interested in conventional, imperative languages in which the chief unit of abstraction is the procedure. We assume that procedure bodies consist of a statement sequence, and may

*Queensland University of Technology, Brisbane, Australia

have an exception handler syntactically associated with the body. When such a procedure is called, the handler of the procedure becomes the **current handler** and the detection of any exception causes control to transfer to that handler. Such a handler remains current during all of the execution of the procedure, except when control has been passed to another procedure which has its own handler. When a procedure with a handler returns, then the previously current handler becomes current once again. Any exception which is detected during the execution of a handler causes control to pass to the dynamically enclosing handler, thus avoiding exception-induced looping.

Our concern in this paper is the implementation of exception handling mechanisms of this kind. As a goal, we would like to entirely eliminate any overhead involved in making particular handlers current, and then removing them. If we are able to achieve this, then a program equipped with exception handlers should run no slower than the same program without, provided that no exceptions are raised during execution. We are even prepared to pay a steep price for the handling of exceptions, provided that it is only the guilty executions which pay, and the innocent get away freely.

THROWING AND CATCHING

The process of raising an exception and having control transfer to the code of the current handler corresponds to a non-local transfer of control. It is very often the case that the procedure in which the exception is detected is different to the procedure which is associated with the currently active handler. In such cases we say that the procedure which is executing when the exception is detected has **thrown** the exception, while the procedure with the current handler is said to have **caught** the exception. There may be an arbitrarily long chain of activation records in the dynamic chain between thrower and catcher.

It is usual for an exception handler to share the namespace of the procedure with which it is associated. Thus the body of the procedure and the associated exception handler share the same activation record. It follows that between the throwing of the exception by the raiser, and the catching by the handler, zero or more activation records may need to be removed from the dynamic chain. In the case that the exception handling model does not provide for direct resumption of the removed contexts, these activation records may be discarded, and we speak of *unwinding the stack* to reach the activation record associated with the handler.

When control reaches the handler, certain aspects of the machine state must have been restored to correspond to the environment of the procedure associated with the handler, that is, the environment of the catcher. With conventional, stack-based runtime organization, we would expect the stack and frame pointers of the catcher to be restored, along with the contents of the display vector, if present.

If any of the possible handler responses may lead to a normal return from the catcher, then it is also necessary to restore all callee-saves registers. If the thrower and catcher are different procedures, then it is possible that the thrower, or indeed any procedure at an intermediate position on the chain may have saved registers which are not saved by the catcher. Unless these registers are restored, the catcher will be unable to meet its obligation to return with **all** callee-saves registers unmodified.

Clearly, a possible implementation would be to save all the components of the state which we have mentioned, in a suitable data structure at entry to each procedure with an exception handler. When an exception is thrown, the runtime system must simply restore the state from the state record. The effect is as if a couroutine transfer has taken place to the handler, with the saved state being analogous to a coroutine state vector. Such a heavyweight saving and restoring of state is the most common way to implement the non-local transfer of control which exception handling requires. This is effectively how all exception handlers built on top of the standard *C* library functions `setjmp` and `longjmp` work.

Such a saving and restoring of the state is much too time consuming to meet our goals, so we seek other possibilities. Note that in order to meet our goal we cannot even use a mutable data structure to represent the chain of current handlers, since the linking and unlinking would involve at least some overhead. Instead, we must entirely reconstruct the machine state from static information represented at runtime by immutable data structures.

PREVIOUS EXPERIENCE

There has been little discussion of techniques for implementing exception handling with low overhead in the literature. However some proposals and results are known.

Language CLU

The language *CLU*[1] has exception handlers which may be associated with individual statements, or larger regions of code. In the declaration of procedures, a list of exceptions raised by that procedure are listed. When invoking such a procedure the source code has the possibility of attaching a handler to each of the listed exceptions, and a default handler for all others. In this language exceptions must be caught by the immediate caller of the routine signalling the exception.

Atkinson *et al*[6] proposed two mechanisms for efficiently implementing this model. The first relied on a table of branch addresses embedded in the object code, immediately followed the call instruction. Error free returns from the invoked procedure would jump over the table, while exceptional returns would vector through the table, using the hardware-saved return address as a base address of the table. The procedure throwing the exception would thus adjust the return address as part of the exit epilog, or take a vectored indirect jump after deleting its activation record in the usual way.

The other method, which was preferred for the final implementations of *CLU*, used a table of handlers. The table contained a pair of instruction address values, together with the various handler addresses. In this case, the list of handlers was searched to find a matching handler with a valid address range which enclosed the known return address. By ordering the table in a way which corresponds to the handler nesting in the program, it is possible to ensure that the first matching handler is the valid one.

Modula-2+

The goals in providing exception handling in *Modula-2+*[2] are directly comparable those stated here. In this language, statement sequences could be enclosed in a *TRY, EXCEPT* blocks. Any exceptions which arose during the execution of the statement sequence in the

TRY block caused control to transfer to the start of the statement sequence in the *EXCEPT* block.

The task of the exception handling implementation for this language, was to finalize enclosed dynamic contexts, unwind the stack and restore registers, before transferring control to the *EXCEPT* clause. After the handler had executed, control passed to the statement immediately following the protected code, in the normal way.

It seems probable that the implementation of this language solved a similar problem to that which we posed as our goal. Unfortunately, no implementation details seem to have been published.

Proposals for C++

Koenig and Stroustrup[7] outline a method for low-overhead exception handling in C++, based on the ideas of Liskov and Rovner. In essence, a table maps code address values to program construction state information. In effect, each table entry is a a triple, containing a code address range, and an address to which control is transferred should an exception occur in that range. The actions associated with each code range include such things as the invocation of object destructor routines.

It was proposed that when an exception occurred, a dispatcher in the runtime system would unwind the stack incrementally, consulting the table in order to find an appropriate destructor strategy to use at each step. It is known[8] that these authors did not actually implement this proposal.

UNWINDING THE STACK

The common element in all of the proposals to provide low-overhead exception handling is the incremental restoration of state, as activation records are removed from the dynamic chain one-by-one. In the case of *CLU* this involves the explicit cooperation of each routine, since each procedure in the dynamic chain must explicitly provide for the propagation of the exception to the caller.

In the other two cases, in principle, the unwinding may include processing of activation records of procedures which have no mention of exception handling in their source text, and need not even have the names of the raised exceptions in their scope.

As an example, suppose we have the dynamic situation depicted in figure 1. Procedure A has called procedure B, which has an exception handler. Procedure B has called C, which has called procedure D, which has raised an exception. In this figure we have assumed a conventional stack organisation, with an explicit stack pointer (sp), and frame pointer (fp). In this case, the stack grows downward in the address space.

It is necessary for the stack frames of D and C to be removed, and any state restoration associated with their return to be executed. Such state restoration is inherently inefficient, as part of the state restored by one step may be overwritten by later restorations. For example, procedures D and C may have saved the same callee-saves registers, and during unwinding the values restored by the unwinding of the D frame will be promptly overwritten by the unwinding of the C frame.

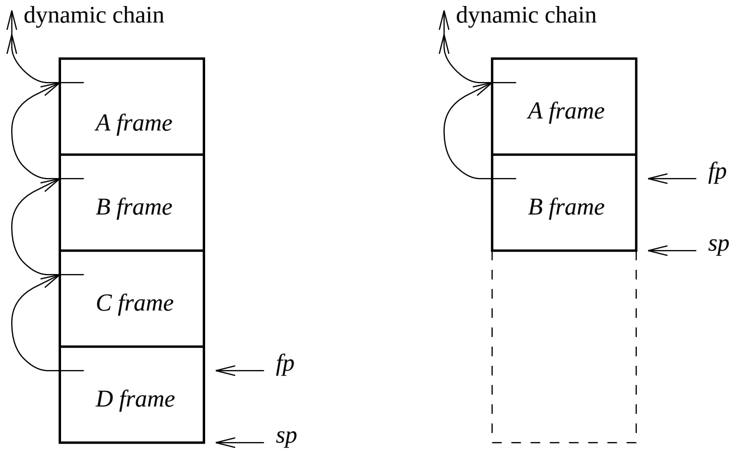

Figure 1: Exception handling example. Procedure D throws, procedure B catches

The essential issues for implementation are —

- to encode the state restoration actions associated with each activation record
- to map from activation records to the unwinding information associated with that record
- to follow the dynamic chain in search of a record with a handler

Of course, every program during non-exceptional execution performs exactly such an incremental state restoration, as procedures return. In this non-exceptional case the state restoration actions are encoded in the epilog of the procedure, and the data of the stack mark. No mapping from activation record to epilog is necessary, since the normal flow of control ensures that the executing code is always associated with the current activation record. Likewise, following the dynamic chain is straightforward. The stack pointer is adjusted by a constant stored in the epilog, while the frame pointer and instruction pointer are adjusted from information stored in the stack mark of the discarded frame.

IMPLEMENTATION

We implemented the exception handling model specified in the draft international standard for *Modula-2*[9]. This proposal is intertwined with another language extension which provides for the finalisation of module bodies. We do not discuss the interaction of the exception handling model and finalisation here.

The **gardens point modula-2** compiler frontend was modified to accept the new syntax. A minor extension was made to the intermediate form[10] used by all the gardens point compilers, providing keyword primitives for marking the start and end of the code region protected by an exception handler, and the start of the handler itself. The code generator used for the experiment targets the *Intel iap486* architecture, under *System V release 4.2*. This particular backend has a conventional stack frame structure, and reserves a register for use as a frame pointer. The runtime organisation uses a display vector for uplevel addressing. This particular backend was chosen as the most straightforward to experiment with. Most of our backends do not waste a register as a frame pointer, even on the Intel architecture. The additional issues raised by systems without frame pointers are discussed later.

We resolved the issues enumerated in the previous section in the following ways.

Encoding the state restoration actions

We considered encoding the state restoration actions in a compact table form, and using a procedure in the runtime support system to interpret this table. However, we finally settled on the emission of inline code which we call the **dummy epilog**. Every procedure has a dummy epilog, even if it has no exception handler. The code generator produces both epilogs, the real and the dummy, based on the the same attributes. Both epilogs may invoke other finalisation actions.

The dummy epilog has inline code to restore callee-saves registers, restore a display element value, if necessary, and to discard the stack frame. In our case, it was a design constraint that if the stack was unwound all the way to the base of the stack, thus invoking the "default handler", a meaningful core dump should be produced for post-mortem debugging. This meant that the stack had to be left intact in cases where no handler was found on the dynamic chain. Thus the unwinding of the stack is always performed in terms of dummy frame, instruction and stack pointers. Once a frame with a handler is found, the real pointers are updated, thus irrevocably discarding the stack information of the unwound procedures.

Mapping from frames to code

As the stack is unwound, it is necessary to map from each frame to the code associated with each frame. Initially, we know the instruction pointer value of the thrower. At each subsequent step we retrieve a return address from the discarded frame. These code addresses allow us to discover the identity of the procedure, and find the procedure descriptor belonging to that procedure. The descriptor contains all required information about the frame. There are two cases, those frames which have both a dummy epilog and an exception handler, and those with just a dummy epilog. We produce tables, each element of which logically[1] contains four data. The format of these procedure descriptors is shown in figure 2, for each case.

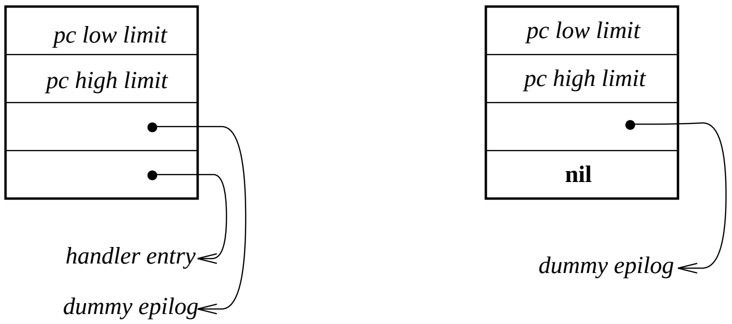

Figure 2: Procedure descriptors for procedures with and without handlers

[1]The actual implementation uses some optimisations which depend on the detailed layout of the program text, together with a tag system to reduce the space usage slightly.

Note that all procedures must have such a procedure descriptor available at runtime. Every procedure must also have a dummy epilog compiled for it, and entered into the descriptor as a symbolic address.

Following the chain

At the end of each dummy epilog, there is code which adjusts the dummy frame and stack pointers, by "removing" the current stack frame. The return address in the discarded stack frame supplies the key which allows the identity of the caller to be determined.

It would have been possible to implement stack frames so that each would contain an explicit reference to the associated procedure descriptor. Currently the stack mark of each activation record contains the return address and the previous frame pointer. The additional pointer would have added just one extra machine word to this structure, but would have required the execution of an additional instruction to store this value in the frame during the entry prolog of every procedure. In the event, we decided to not allow even a single instruction of overhead in the prolog.

Instead, we implemented the table element structure shown in figure 2. We search these tables to find the procedure descriptor which matches the return address of the last frame, so as to determine the identity of the next frame.

The logic of the unwinding algorithm may be symbolically represented as

```
dummyFP := fp; dummySP := sp; dummyIP := ip;
find the procedure descriptor corresponding to the current dummyIp;
while this frame does not have a handler do begin
   execute the dummy epilog to restore state;
   update the dummy pointers as specified;
   find the procedure descriptor corresponding to the current dummyIp;
end;
if discovered handler is the default then
   abort with coredump;
else begin
  fp := dummyFP; sp := dummySP;
   jump to handler start address;
end
```

The work is all in the finding of the procedure descriptors.

Creating the address map

Each compilation unit is compiled to assembly language, which contains data declarations for the procedure descriptors. This data area has a generated name synthetically related to the module name. The addresses in these tables will be resolved at link-edit time.

Modula-2 provides for automatic invocation of the initialisation code of all modules in a program, with the order of initialisation determined by the importation graph of the program. We implement separate compilation of modules, and linking is controlled by the **build** tool. Build reads import lists from the reference file of each module, and constructs the imports directed graph. A topological sort on this graph determines a valid initialisation order, initialisation code is emitted, and the system linker invoked. We augmented the build tool to construct a global address map, from the module list determined from the imports graph. At link time, we therefore determine initialisation order, and construct the first level of a two-level address map. The second level of mapping is already present in the compiled modules.

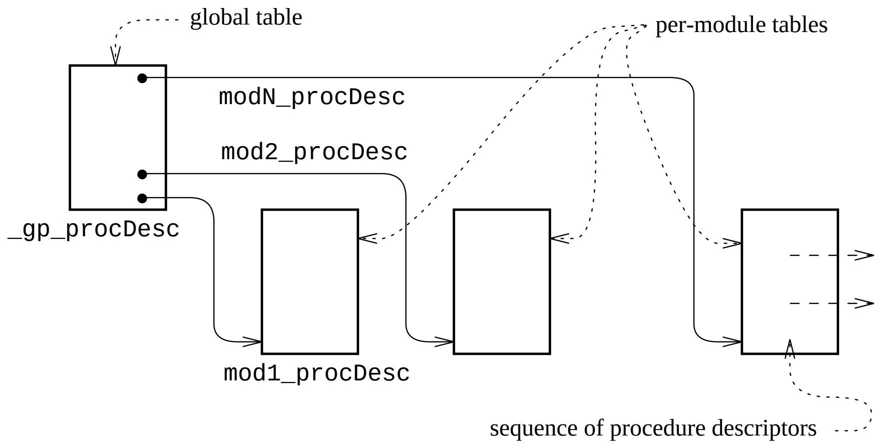

Figure 3: Two-level descriptor map, with global and per-module tables

Figure 3 shows the address map structure. A global table placed at a known symbolic address contains a nil-terminated array of pointers to the per-module tables. Each per-module table starts with a maximum address, followed by an array of the procedure descriptors shown in figure 2. The address of each per-module table is an external symbol which is resolved at link-edit time, while the internal addresses in each procedure descriptor are not necessarily global symbols, and may be resolved to an offset at assembly time.

The lookup algorithm, in order to find each descriptor, first searches the global table to find the per-module table which contains the current code address. The procedure descriptor is then found by a linear search through the chosen descriptor array.

PERFORMANCE

We were successful in achieving our stated goal, since the resulting exception handling system does not execute a single additional instruction during program executions which do not raise exceptions. We discuss whether this is the same thing as having no performance penalty in the final section.

Having deliberately sacrificed exception handling speed in the interests of zero overhead, we were interested to quantify the extent of the sacrifice. We measured the time taken to handle 10^5 exceptions, with the exception raised by a recursive procedure. We varied the depth of recursion at which the exception occurred, so as to measure the differential time penalty associated with each additional level of stack unwinding. The time taken to remove a stack frame is essentially independent of the size of the frame, but depends on the length of the search in the hierarchical tables of figure 3. For our example, the global table search length was four, and the per-module search length was ten.

We measured the times using a 33MHz *Intel*-486DX. The time taken per additional stack frame was consistently $20\mu S$. This corresponds to about 400–500 instructions, implying a plausible figure of 25 instructions per table element lookup, with an additional few instructions to update the dummy stack-state pointers for each frame.

Between the detection of an exception and the start of the unwinding certain housekeeping must take place. In the case of *ISO Modula-2*, data is created which is used to identify the exception identity and source. The time taken to perform this startup depends on the source of the exception in a way which confounded our initial expectations. The lowest time was required by language traps such as range errors. These are detected by compiler generated code, and call directly to a runtime support routine.

Traps which are detected by the hardware, and passed to the program as operating system signals take slightly longer. In the case of division-by-zero errors, the additional time was measured as $40\mu S$ per exception. We think this is a surprisingly moderate overhead, given that there are two context switches and a trap dispatcher execution included.

The explicit raising of an exception using *ISO Modula-2*'s *EXCEPTIONS.RAISE* procedure, was unexpectedly time consuming. We measured the additional time at $330\mu S$ per exception. It appears that much of this time is taken up by housekeeping operations in the exceptions module. In particular, the raise procedure takes a value-mode string argument, which is reported to the user if the exception finally causes program termination. With our present design, this string argument is copied multiple times, as various procedures in the exception handling library call each other. System-defined exception sources have fixed error message strings encoded in the runtime system, and escape this overhead entirely.

DISCUSSION

There are several interesting points which arise from the implementation experience described here. We have claimed that the dynamic setting and removal of handlers has zero overhead. However, this may not be quite the same thing as having no performance penalty.

In a typical application we might have code of the general form —

```
PROCEDURE Foo(p : T);
  VAR l : T;
BEGIN
  LOOP
     body of loop raises exception;
  END;
  ...
EXCEPT (* handler starts here *)
   handler body accesses p and l;
END Foo;
```

The question is, how do we ensure that the definitions of p and l in the body of the procedure will reach uses of these local variables in the handler body. In particular, if either of these variables is promoted to a register, then the value may not be preserved by all of our state restoration efforts. The problem is that graph colouring register allocators will routinely allocate different live ranges of the same "register variable" to different registers. Since the presence of exception handling introduces control flow (even interprocedural control flow) which is not represented in the control flow graph, there will be definition-use pairs which we are unable to discover. Since most of our code generators use graph colouring allocators, we are forced to allocate all those local variables which are used in both the body and the handler to memory. The semantics are precisely those required for *ANSI C* 'volatile' variables. A simpler but even more draconian strategy would be to turn off register variables for every procedure which has an exception handler.

There is thus a small performance penalty associated with the use of zero overhead exception handling, caused by the foregoing of optimisation opportunities. This penalty is likely to be more significant for *RISC* machine architectures, which rely to a greater extent on the use of register variables.

The possibility of including a pointer to the procedure descriptor in each stack mark merits some further discussion. Including such a datum would remove the main inefficiency involved in the search for the current handler. It may be argued that the cost of one or two instructions in each prolog may very well be justified to provide better handling speed.

The implementation of such a scheme in the case of stack conventions which do not use a frame pointer is quite constrained. A typical stack frame in such an organisation is shown in figure 4

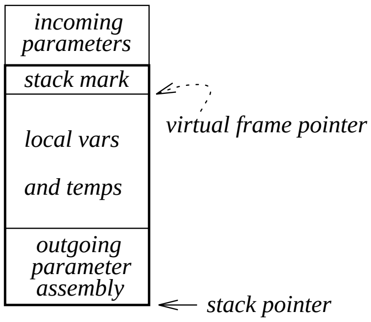

Figure 4: Typical stack frame, with virtual frame pointer

In such conventions the stack frame size is fixed, and known to the compiler at code generation time. Accesses to local data are adjusted at compile time to be made relative to the stack pointer, rather than a frame pointer. The stack mark, if it is not empty, only contains a return address, and is only needed during procedure exit.

With such stack frames we cannot place the descriptor pointer in the stack mark, since the unwinder cannot locate the stack mark unless it already knows the stack frame size. Since the frame size is stored in the descriptor, the prohibition follows.

The only possibility seems to be to place the pointer at a fixed offset from the frame pointer. Since the size of the outgoing parameter assembly area is equally unknown to the unwinder, it follows that the pointer can only be at the very top of the stack, above the parameter assembly area. This convention seems possible, causing a minor adjustment to the parameter assembly offsets. We have not implemented this scheme however.

Finally, it is important to recognise that the methods which we have discussed cannot reliably deal with exceptions which originate from asynchronous events. The problem is that the process of unwinding the stack implicitly assumes that procedure prologs and epilogs are atomic events. If an asynchronous event occurs midway during the saving of callee-saves registers, as an example, then it will be incorrect to execute the dummy epilog, and equally incorrect to not execute it. For this reason, we take the low address of the protected region of a procedure as a point immediately after the saving of registers. The code which makes callee-copies of large value parameters lies within the protected region, thus providing correct exception handling in the event of a segment violation due to invalid parameter data. However, if a kill signal is received during a procedure prolog, the unwinder will be unable to find a matching procedure descriptor, and will go directly to the default handler.

The exception handling technique described here seems to be able to be adapted without major change to our other code generators. We expect to implement essentially the same system on the reduced instruction set computers which form the remainder of our target architectures.

References

[1] B. H. Liskov and A. Snyder, 'Exception Handling in CLU', *IEEE Transactions on Software Engineering*, **SE-5**, (6), 546–558, (1979)

[2] P. Rovner, 'Extending Modula-2 to Build Large, Integrated Systems', *IEEE Software*, **3**, (6), 47–57, (1986)

[3] F. Cristian, 'Correct and Robust Programs', *IEEE Transactions on Software Engineering*, **SE-10**, (2), 163–174, (1984)

[4] F. Cristian, 'Exception Handling', in *Dependability of Resilient Computers*, Ed T. Anderson, BSP Professional Books, Blackwell Scientific Publications. Ch4, 68–97, (1989)

[5] S. J. Drew and K. J. Gough, 'Exception Handling: Expecting the Unexpected', *Computer Languages*, **32**, (8), 69–87, (1994)

[6] R. R. Atkinson, B. H. Liskov and R. W. Schiefler, 'Aspects of Implementing CLU', *Proceedings of the Annual Conference of the ACM*, 123–129, (1978)

[7] A. Koenig and B. Stroustrup, 'Exception Handling for C++ (revised)', *Proceedings of the USENIX C++ Conference*, 149–176, (1990)

[8] B. Stroustrup, Personal communication with S. Drew, December 1994

[9] ISO, 'Modula-2 Draft Standard', ISO/IEC DIS 10514:1994

[10] K. J. Gough, 'DCode Reference Manual and Report', QUT report 1994, available from internet site `pluto.fit.qut.edu.au` in directory `/pub`