Reprinted from *Proceedings of the 21st ACM Symposium on the Principles of Programming Languages*, January 1994.

Correctness of Trap-Based Breakpoint Implementations

Norman Ramsey
Bell Communications Research
445 South Street, Morristown, NJ 07960
norman@bellcore.com

Abstract

It is common for debuggers to implement breakpoints by a combination of planting traps and single stepping. When the target program contains multiple threads of execution, a debugger that is not carefully implemented may miss breakpoints. This paper gives a formal model of a breakpoint in a two-threaded program. The model describes correct and incorrect breakpoint implementations. Automatic search of the model's state space shows that the correct implementation never misses a breakpoint. A similar search finds an execution for which the incorrect implementation does miss a breakpoint. The results apply even to debuggers like `dbx` and `gdb`, which are apparently for single-threaded programs; when the user evaluates an expression containing function calls, the debugger executes the call in the target address space, in effect creating a new thread.

1 Introduction

A debugger runs as a coroutine with its target program. A breakpoint at target instruction I transfers control from the target to the debugger whenever control reaches I. When the debugger gets control, it may take such actions as evaluating a condition, incrementing a counter, or simply asking the user for instructions. Eventually the debugger returns control to the target, which resumes execution at I. At that time, I must be executed once, but subsequent attempts to execute I must return control to the debugger.

Implementors can choose how to manage the transfers of control. To get control at an instruction I, a debugger can overwrite I with a trap instruction, then handle the resulting trap (Caswell and Black 1990), or it can overwrite I with an instruction that branches to debugging code (Digital 1975). To resume execution, there are more choices. A debugger can return the overwritten instruction to memory, execute it by single stepping the target machine, and re-plant the breakpoint (Caswell and Black 1990). Single stepping can be avoided by transforming the overwritten instruction so that it can be correctly executed out of line (Digital 1975; Kessler 1990). Finally, some machines have special hardware that supports resumption after a break instruction (Bruegge 1985).

This paper exposes a potential pitfall in the implementation of breakpoints based on trapping and single-stepping. Single stepping means arranging that the target machine will trap again immediately after executing I. On some machines it can be implemented only by planting traps at instructions that might be executed immediately after I; these instructions are I's *follow set*. If these traps are planted at the wrong time, and if the target has more than one thread of control, the debugger could miss a breakpoint.

Even a traditional single-threaded debugger must avoid this pitfall if it evaluates expressions at the source level. Expression evaluation includes calls to user-defined procedures, and these procedures run in the target address space. To run such procedures, the debugger must in effect create a second thread. If such a procedure hits a breakpoint and the expression evaluation is abandoned, the debugger may miss a subsequent arrival at that breakpoint.

This paper makes several contributions. It provides an abstraction and formalization of a breakpoint technique commonly used with compiled forms of imperative programming languages. Using the formalism, the paper shows that unrestricted context switching can make a debugger miss a breakpoint. It gives restrictions on context switching that prevent the error; these restrictions are shown sufficient by machine checking of all possible executions. Finally, the paper gives an un-

POPL 94- 1/94, Portland Oregon, USA

Debugger	**Target**
	trap at I
catch trap at I	
detect breakpoint	
restore I	
single-step I	
	I refers to a bad address
catch fault	
user corrects bad address value	
resume execution	
	I executes
	single-step completes and traps
catch trap at completion	
re-write trap at I	
resume execution	
	target continues

Figure 1: Address fault in the middle of a breakpoint (time flows downward)

usual application of protocol-validation techniques to a programming-language problem.

2 Breakpoints and events

In a trap-based implementation, breakpoint handling begins when a target thread hits the trap written over instruction I. The description above says that I is returned to memory and the machine is single-stepped, after which I can be replaced with a trap. This description is oversimple, because I need not execute successfully. For example, it may refer to an invalid address, as shown in Figure 1 (time flows from top to bottom). A practical breakpoint implementation cannot rely on executing a simple sequence of (a) remove trap, (b) single step, (c) replace trap; it must be prepared for other events to be inserted into the sequence, and it must create handlers that respond appropriately to those events.

Two handlers are relevant to the breakpoint implementation. One handles the trap that indicates a thread's attempt to execute instruction I, and the other handles the notification that the execution of I (the single step) has completed successfully. These handlers appear in Figure 1 as the first and third actions of the debugger. The second action is that of a handler responding to the invalid-address event; it is not part of the breakpoint implementation.

Before letting the target execute I, the first handler must arrange for an event to occur when I's execution completes. The usual choice is a trap event, which can be arranged either by setting a trace bit in the target processor (if available) or by planting traps at I's follow set. Once the bit is set or the traps are planted, the execution of the target thread can be resumed. The second handler, when it sees the trap at the completion of I, undoes the trace bit or the traps at the follow set and rewrites a trap over I.

This implementation is incorrect in the presence of multiple threads of execution. Figure 2 shows an execution involving a debugger and two target threads in which both threads go through the breakpoint, but only one is detected. As shown in the next section, the problem arises because both threads are permitted to execute after I is restored. The solution is to prevent the execution of other threads while the thread that trapped executes I.

The problem can arise in practice even in a single-threaded debugger. When the thread traps at I, the debugger may evaluate an expression before resuming execution. If a procedure call is needed to evaluate the expression, the debugger must, in effect, create a second target thread to call the procedure. If this second thread hits the breakpoint at I, it will not be detected. An early version of the author's debugger `ldb` demonstrated this problem (Ramsey 1992b). The problem cannot be demonstrated with `dbx` (Linton 1990) or `gdb` (Stallman and Pesch 1991) because neither is capable of recovering if a procedure call hits a breakpoint while the main target thread is stopped at a breakpoint.

3 Model using communicating sequential processes

The debugger, target, and breakpoint implementation are modeled as communicating sequential processes. The breakpoint implementation is to notify the debugger every time a thread successfully executes instruction I; to that end it can plant and remove traps, and it can suppress or permit context switching. The model is abstract, hiding details. It does not matter whether the debugger and target reside in the same address space or in different address spaces, nor does it matter whether the breakpoint implementation resides in the target, the operating system, or the debugger. It does not matter whether the target processor has a trace mode.

The model is expressed in PROMELA, a formalism whose original purpose was to model network protocols (Holzmann 1991). The correctness condition is given by embedded assertions, which state that the number of executions of I is equal to the number of notifications of the debugger. The advantage of using PROMELA is that it comes with a checker that searches a model's state space and checks for deadlocks, unreachable states, and violated assertions.

Debugger	thread 0	thread 1
	trap at I	
catch trap at I		
detect breakpoint		
restore I		
plant trap at I's successor		
resume execution		
	execute I	
		execute I
		trap at I's successor
catch trap at I's successor		
note completion of breakpoint		
restore I's successor		
plant trap at I		
resume execution		
		execute I's successor
	execute I's successor	
	continue executing	continue executing

Figure 2: Execution sequence showing missed breakpoint

PROMELA models need not be deterministic. When a choice is nondeterministic, the checker explores all alternatives, so it will find an execution leading to a violated assertion if one exists.

Syntactically, PROMELA models resemble C programs. They use `!` and `?` operators to send and receive messages over "channels," *à la* CSP (Hoare 1978). Although PROMELA permits models in which messages are passed asynchronously using buffered channels, the model presented here uses only unbuffered channels; every send and receive is a synchronization point, as in CSP.

This paper does not just describe the model; it contains the model. The `noweb` system (Ramsey 1992a) for "literate programming" (Knuth 1984) extracts both the paper and the model from the same source. The source contains prose interleaved with definitions of named code "chunks." Definitions are numbered consecutively with bold numerals. Code chunks contain source code and references to other code chunks. The code chunks appear in an order suited to explanation, not necessarily in the order required by the PROMELA language. Chunk names appear italicized within angle brackets, e.g., ⟨*declarations* 1⟩. The "1" is the number of the chunk's first definition.

3.1 Processes of the model

Five processes are used to model the interaction of debugger, target, and breakpoint implementation. Two processes model threads of control in the target, one models the debugger, and one the breakpoint implementation. The fifth process models the CPU, which advances the program counter, notifies the breakpoint implementation when it hits a trap, and notifies the active target thread when it successfully executes an instruction. The processes and communications between them are shown in Figure 3. Using only two threads keeps the state space small; a single bit suffices to identify a thread. A small state space is necessary for exhaustive search to be practical.

1. ⟨*declarations* 1⟩≡
```
#define NTHREADS 2
#define threadid bit
```
This definition is continued in chunks 2–5, 8, 10, 18, 22, and 26.
This code is used in chunk 29.

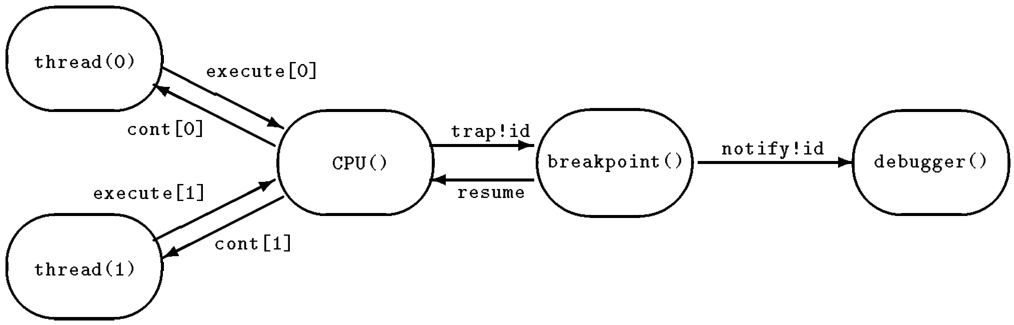

Figure 3: Processes used in the model

The labels on the arcs in Figure 3 designate the following channels:

2. ⟨*declarations* 1⟩+≡

```
chan execute[NTHREADS] = [0] of {bit};
chan cont    [NTHREADS] = [0] of {bit};
chan trap               = [0] of {threadid};
chan resume             = [0] of {bit};
chan notify             = [0] of {threadid};
```

Defines:
`cont`, used in chunks 6 and 12.
`execute`, used in chunks 6 and 12.
`notify`, used in chunks 13, 24, and 25.
`resume`, used in chunks 6, 24, and 25.
`trap`, used in chunks 6, 24, and 25.

`[0]` is the size of the buffer associated with the channel; these channels have buffers of length zero and are therefore synchronous. `{bit}` or `{byte}` shows the data included in a message.[1] The `execute` channels, one per thread, are used by the threads to ask the CPU to attempt to execute the instruction. If the attempt is successful, the CPU adjusts the PC and replies on the corresponding `cont` channel. If the attempt traps, the CPU sends the unique `id` of the trapping thread on the `trap` channel to the breakpoint implementation. The breakpoint implementation adjusts traps and tells the CPU to resume execution by sending on the `resume` channel. If the trap indicates a new breakpoint event, the breakpoint implementation notifies the debugger that a breakpoint occurred by sending the `id` of the breakpointing thread on the `notify` channel.

3.2 Modeling the program counter and execution

Also to reduce the size of the state space, the model has only one breakpoint. Modeling all possible values of the program counter is too expensive, but the values partition naturally according to the instruction the program counter refers to:

`Break`	the breakpoint itself (instruction I),
`Follow`	the instruction(s) that can follow I,
`Outside`	other instructions.

The three sets are modeled by the following constants.

4. ⟨*declarations* 1⟩+≡

```
#define Break   0  /* pc at I */
#define Follow  1  /* pc in I's follow set */
#define Outside 2  /* all other pc's */
#define NPCS    3  /* no. of distinct pc's */
```

Defines:
`Break`, used in chunks 7, 12, 14, 17–21, 24, and 25.
`Follow`, used in chunks 7, 17, 20, 21, 24, and 25.
`Outside`, used in chunks 7, 9, and 12.

The ability to plant traps is modeled by the array `trapped`, which records whether a trap instruction has been stored at a particular location:

5. ⟨*declarations* 1⟩+≡

```
bool trapped[NPCS];
```

Defines:
`trapped`, used in chunks 6, 17, and 19–21.

[1] `resume` and the `cont` and `execute` channels are used only for synchronization, but PROMELA does not permit a message without data, so these channels carry the one-bit value `x`, which is always ignored.

3. ⟨*declarations* 1⟩+≡

```
bit x;       /* sent/received on synch channels */
```

The CPU repeats the following steps.

1. Wait for a thread to attempt to execute the instruction at `pc`.
2. If the instruction is a trap, notify the breakpoint implementation. When the CPU is told to resume, `pc` is unchanged.
3. If the instruction is not a trap, advance `pc`.
4. Ask the thread to continue executing.

There is only one debugger, but there are multiple threads, and each one has its own `pc` and its own communication with the CPU. When the CPU notifies the debugger of a trap, it identifies the trapping thread.

6. ⟨*proctypes* 6⟩≡
```
proctype CPU() {
  threadid id = 0;
  do
  :: execute[id]?x ->
     if
     ::  trapped[pc[id]] -> trap!id ; resume?x
     :: !trapped[pc[id]] -> ⟨advance pc[id] 7⟩
     fi;
     cont[id]!x;
     ⟨possible context switch (change of id) 27⟩
  od
}
```
Defines:
  `CPU`, used in chunk 29.
Uses `cont` 2, `execute` 2, `resume` 2, `trap` 2, and `trapped` 5.
This definition is continued in chunks 12, 13, 24, and 25.
This code is used in chunk 29.

A PROMELA `proctype` defines the actions taken by a PROMELA process. `c?x` receives the value `x` on channel `c`; `c!x` sends. `do`, `if`, and `::` are comparable to the iteration, alternation, and bar ($\Box$) commands from Dijkstra's (1976) calculus of guarded commands. Their semantics differ when all guards are false; the PROMELA `if` and `do` *block* waiting for a guard to become true, but Dijsktra's iteration command terminates, and his alternation command aborts. In this example, the guards make the `if` command deterministic.

The only significant source of nondeterminism in the model is represented by the chunk ⟨*possible context switch (change of* `id`*)* 27⟩. If `id` can change after every attempt to execute an instruction, instructions from the two threads can be interleaved arbitrarily. Section 3.4 shows the part of the model that handles context switching; `id` can change only if such change is permitted by the breakpoint implementation.

Since the program counter is an abstraction, advancing it does not mean incrementing it. By the definition of follow set, a successful execution at `Break` is guaranteed to be followed by an attempt to execute `Follow`; aside from that, any instruction can follow any other.

7. ⟨*advance* `pc[id]` 7⟩≡
```
if
:: pc[id] == Break -> pc[id] = Follow
:: pc[id] != Break ->
   /* any instruction can be next */
   if
   :: pc[id] = Outside
   :: pc[id] = Break
   :: pc[id] = Follow
   fi
fi
```
Uses `Break` 4, `Follow` 4, and `Outside` 4.
This code is used in chunk 6.

Because the second `if` statement has no guards, an alternative is chosen nondeterministically. This nondeterminism is not essential to the model, and it does not affect the correctness of the breakpoint implementation; it exists only to abstract away from the exact sequence of instructions executed by a particular thread.

All threads begin execution outside the breakpoint.

8. ⟨*declarations* 1⟩+≡
```
byte pc[NTHREADS];
```

9. ⟨*initialize thread* `id`*'s data* 9⟩≡
```
pc[id] = Outside;
```
Uses `Outside` 4.
This definition is continued in chunks 11 and 23.
This code is used in chunk 30.

3.3 Counting events

A correct breakpoint implementation guarantees that the debugger is notified exactly once for every trip a target thread takes through the breakpoint. The counter `threadcount[id]` counts how many times thread `id` has executed the breakpoint. The counter `notifycount[id]` counts how many times the debugger has been notified that thread `id` executed the breakpoint.

10. ⟨*declarations* 1⟩+≡
```
byte threadcount[NTHREADS];
byte notifycount[NTHREADS];
```
Defines:
  `notifycount`, used in chunks 11, 12, and 16.
  `threadcount`, used in chunks 11, 12, and 15.

11. ⟨*initialize thread* `id`*'s data* 9⟩+≡
```
threadcount[id] = 0;
notifycount[id] = 0;
```
Uses `notifycount` 10 and `threadcount` 10.

The thread model maintains threadcount[]. If the program counter is Break, execution is attempted, and the program counter is no longer Break, then instruction I has been executed and the thread counter must be incremented. If the program counter remains Break after the attempt, the attempted execution failed, and the counter should not be incremented.[2] The thread model also contains an embedded assertion stating that, unless the thread is in the middle of a breakpoint, the thread and debugger counts are the same:

12. ⟨*proctypes* 6⟩+≡

```
proctype thread(threadid id) {
  do
  :: if
     :: pc[id] == Break ->
          execute[id]!x; cont[id]?x;
          ⟨if pc ≠ Break, increment thread count 14⟩
     :: pc[id] != Break ->
          execute[id]!x; cont[id]?x
     fi;
     assert(pc[id] != Outside ||
            threadcount[id] == notifycount[id])
  od
}
```

Defines:
thread, used in chunk 29.
Uses Break 4, cont 2, execute 2, notifycount 10, Outside 4, and threadcount 10.

The debugger, when notified that thread id has hit the breakpoint, increments notifycount[id].

13. ⟨*proctypes* 6⟩+≡

```
proctype debugger() {
  threadid id;
  do
  :: atomic {
       notify?id -> ⟨increment notifycount[id] 16⟩
     }
  od
}
```

Defines:
debugger, used in chunk 29.
Uses notify 2.

[2] I discount the possibility that instruction I branches to itself. This possibility breaks the model's assumption that Follow is distinct from Break. It also breaks real breakpoint implementations that rely on adjustment of traps by software; only implementations that use a hardware trace bit can handle such instructions. This possibility is unimportant in practice because the machines that have no trace bit are RISC machines, on which it is not useful to write programs containing instructions that branch to themselves.

The PROMELA atomic keyword groups actions into a single atomic action. When the debugger is notified, it atomically increments notifycount[id]. Without atomic, it might delay incrementing the counter until the thread left the breakpoint, which would lead to a spurious violation of the assertion above.

A thread has successfully executed Break if the pc has changed:

14. ⟨*if pc* ≠ Break*, increment thread count* 14⟩≡

```
if
:: pc[id] != Break ->
     ⟨increment threadcount[id] 15⟩
:: pc[id] == Break -> skip
fi
```

Uses Break 4.
This code is used in chunk 12.

Restricting the values of the counters to be in the range 0..3 keeps the state space small.

15. ⟨*increment* threadcount[id] 15⟩≡

```
threadcount[id] = (threadcount[id] + 1) % 4
```

Uses threadcount 10.
This code is used in chunk 14.

16. ⟨*increment* notifycount[id] 16⟩≡

```
notifycount[id] = (notifycount[id] + 1) % 4
```

Uses notifycount 10.
This code is used in chunk 13.

3.4 Implementing the breakpoint

The standard description of trap-based breakpoint implementations refers to single stepping. To set a breakpoint at I, plant a trap at I. When the target program hits the trap, notify the debugger of a breakpoint event. To resume execution after the breakpoint, restore the original instruction to I, single step the machine to execute just the instruction at I, and once again plant a trap at I and continue execution.

This model eliminates single stepping entirely, working directly with trap instructions and a follow set (modeled by Follow). It does not, however, preclude the use of hardware single stepping. One of the operations in the model is planting traps at the locations in the follow set of an instruction. This operation can be implemented either by computing the follow set and planting actual traps, or by setting a trace bit on a machine with hardware single stepping.

An active breakpoint is trapped either on the instruction of the breakpoint itself or the instructions in its follow set. The variable breakstate keeps track of which state it is in, with the following invariant.

17. ⟨*invariant* 17⟩≡

```
   breakstate == Break &&
   trapped[Break] == 1 && trapped[Follow] == 0
|| breakstate == Follow &&
   trapped[Break] == 0 && trapped[Follow] == 1
```

Uses Break 4, breakstate 18, Follow 4, and trapped 5.
This code is used in chunks 20 and 21.

18. ⟨*declarations* 1⟩+≡
```
byte breakstate = Break;
```
Defines:
`breakstate`, used in chunks 17, 20, 21, 24, and 25.
Uses `Break` 4.

19. ⟨*initialization* 19⟩≡
```
trapped[Break] = 1;
```
Uses `Break` 4 and `trapped` 5.
This code is used in chunk 29.

Changing the state preserves the invariant.

20. ⟨*move traps to* `Break` 20⟩≡
```
atomic {
  trapped[Break] = 1;  trapped[Follow] = 0;
  breakstate = Break;  assert ⟨invariant 17⟩
}
```
Uses `Break` 4, `breakstate` 18, `Follow` 4, and `trapped` 5.
This code is used in chunks 24 and 25.

21. ⟨*move traps to* `Follow` 21⟩≡
```
atomic {
  trapped[Break] = 0;  trapped[Follow] = 1;
  breakstate = Follow; assert ⟨invariant 17⟩
}
```
Uses `Break` 4, `breakstate` 18, `Follow` 4, and `trapped` 5.
This code is used in chunks 24 and 25.

It is necessary to keep track of the state of each thread with respect to the breakpoint. A thread is "in the breakpoint" if it has trapped at `Break`, and it does not "leave the breakpoint" until it traps at `Follow`. Threads are initially outside the breakpoint.

22. ⟨*declarations* 1⟩+≡
```
bit  inbreak[NTHREADS];
```
Defines:
`inbreak`, used in chunks 23–25.

23. ⟨*initialize thread* `id`*'s data* 9⟩+≡
```
inbreak[id] = 0;
```
Uses `inbreak` 22.

The incorrect implementation described in the introduction keeps track of the various states and delivers a breakpoint event at the right time:

24. ⟨*proctypes* 6⟩+≡
```
proctype badbreakpoint() {
  threadid id;

  do
  :: trap?id ->
     if
     :: breakstate == Break ->
        if
        :: !inbreak[id] -> notify!id;
                           inbreak[id] = 1
        ::  inbreak[id] -> skip
                           /* no event */
        fi;
        ⟨move traps to Follow 21⟩
     :: breakstate == Follow ->
        if
        ::  inbreak[id] -> inbreak[id] = 0
        :: !inbreak[id] -> skip
        fi;
        ⟨move traps to Break 20⟩
     fi;
     resume!x
  od
}
```
Uses `Break` 4, `breakstate` 18, `Follow` 4, `inbreak` 22, `notify` 2, `resume` 2, and `trap` 2.

The two cases

`breakstate == Break`

and

`breakstate == Follow`

represent the two handlers described in the introduction. This implementation works correctly with one thread, but with two threads it permits the erroneous execution sequence shown in Figure 2.

To prevent such an occurrence, the processor must not be permitted to change contexts when a thread is in the middle of a breakpoint. If the processor can change contexts only when `noswitch == 0`, then the following breakpoint implementation works correctly.

25. ⟨*proctypes* 6⟩+≡

```
proctype breakpoint() {
  threadid id;

  do
  :: trap?id ->
     if
     :: breakstate == Break ->
        if
        :: !inbreak[id] -> notify!id;
                           inbreak[id] = 1
        ::  inbreak[id] -> assert(0)
        fi;
        noswitch = noswitch + 1;
        ⟨move traps to Follow 21⟩
     :: breakstate == Follow ->
        if
        ::  inbreak[id] -> inbreak[id] = 0
        :: !inbreak[id] -> assert(0)
        fi;
        ⟨move traps to Break 20⟩;
        noswitch = noswitch - 1
     fi;
     resume!x
  od
}
```

Defines:
`breakpoint`, used in chunk 29.
Uses `Break` 4, `breakstate` 18, `Follow` 4, `inbreak` 22, `noswitch` 26, `notify` 2, `resume` 2, and `trap` 2.

When context switching is forbidden, `breakstate` always reflects the information in `inbreak`, and it is possible to take each `skip` from the bad breakpoint implementation and strengthen it to `assert(0)` in the good implementation.

Because there is only one breakpoint, `noswitch` could be a bit, not a counter, but a counter generalizes to multiple breakpoints.

26. ⟨*declarations* 1⟩+≡

```
byte noswitch = 0;
```

Defines:
`noswitch`, used in chunks 25 and 27.

The model's CPU may change threads only when `noswitch` is zero:

27. ⟨*possible context switch (change of* `id`*)* 27⟩≡

```
if
:: noswitch == 0 -> ⟨set id nondeterministically 28⟩
:: noswitch  > 0 -> skip
fi
```

Uses `noswitch` 26.
This code is used in chunk 6.

28. ⟨*set* `id` *nondeterministically* 28⟩≡

```
atomic {
  if
  :: id = 0
  :: id = 1
  fi
}
```

This code is used in chunk 27.

When `noswitch` is nonzero, the nondeterministic choice of `id` models the arbitrary interleaving of instructions that takes place in a multiprocessor, or the arbitrary interleaving that takes place when threads are scheduled pre-emptively on a uniprocessor.

4 Results

For the code shown above to be a proper PROMELA model, the `proctype` declarations must follow the other declarations, and there must be initialization actions that create the five processes and their associated channels.

29. ⟨*PROMELA model* 29⟩≡

```
⟨declarations 1⟩
⟨proctypes 6⟩
init {
  threadid id;
  atomic {
    ⟨initialization 19⟩
    ⟨for 0 ≤ id < NTHREADS, initialize thread id's data 30⟩;
    run thread(0);
    run thread(1);
    run debugger();
    run breakpoint();
    run CPU(2)
  }
}
```

Uses `breakpoint` 25, `CPU` 6, `debugger` 13, and `thread` 12.
This chunk expands to the complete PROMELA model.

30. ⟨for $0 \leq$ `id` $<$ `NTHREADS`, initialize thread `id`'s data 30⟩≡

```
id = 0;
do
:: id < NTHREADS -> ⟨initialize thread id's data 9⟩
   if
   :: id == NTHREADS - 1  -> break
   :: id <  NTHREADS - 1  -> id = id + 1
   fi
od
```

This code is used in chunk 29.

The full model is 169 (narrow) lines long, including both good and bad breakpoint implementations. The PROMELA checker takes about 4.2 seconds on a SPARC 10 to check all possible executions for violations of the embedded assertions. By finding none, it shows that the suppression of context switching in the correct implementation is sufficient to guarantee that all executions of I are reported to the debugger. If the bad breakpoint implementation is used instead of the good one, the checker takes less than a second to find an execution that leads to an assertion violation. Such an execution was used to prepare Figure 2. The result may be stated thus:

> While the memory at I holds the original instruction and not a trap, only the thread that trapped at I may be permitted to execute.

5 Related work

Caswell and Black (1990) describe the implementation of a multithreaded debugger. They mention that they use a breakpoint implementation based on trapping and single-stepping, but they do not identify the pitfall. Redell (1989) alludes to the problem, indicating that careful design is necessary in the debugger's treatment of threads and events. Elsewhere, I have described in detail a breakpoint implementation based on the model presented here (Ramsey 1992b).

Much of the work on breakpoints has focused on performance. Kessler (1990) describes a fast implementation of code breakpoints; Wahbe (1992) describes simulations of four implementations of data breakpoints. In a parallel environment, debugging work can be offloaded onto a second processor (Aral, Gertner, and Schaffer 1989). Alternatively, monitoring and logging can be done by a special-purpose coprocessor (Gorlick 1991). All these approaches use either branch or coprocessor instructions to transfer control from target to debugging code without kernel intervention, avoiding the overhead of trap handling and context switching. The authors with working implementations describe performance improvements of three orders of magnitude over trap-based breakpoints as implemented on Unix.

6 Discussion

Breakpoints may be implemented either in the operating system or in the debugger itself; the choice does not affect the model used here. Although they use similar breakpoint implementations, the Topaz teledebugger puts the breakpoint implementation in the operating system (Redell 1989); `ldb`, the author's Unix teledebugger, puts it in the debugger (Ramsey 1992b). The model assumes it can plant trap instructions in the instruction stream of the target program, and that it will be notified when the target program encounters a trap. The model also suits a machine with a "trace mode" that causes a trap after the execution of every instruction.

The trap-based implementation of breakpoints is usually explained in terms of instruction-level single stepping. This explanation misleadingly suggests that breakpoints can be implemented using simple, sequential code. In fact, the implementation must be written in a kind of continuation-passing style, using event handlers to match continuations with events. Thinking in terms of traps at follow sets makes it easier to understand the real implementation. It also clarifies the relationship between an implementation that uses only traps and one that uses a hardware trace mode; planting or suspending traps in a follow set is equivalent to setting or clearing a trace bit in a program status word.

The model forbids context switching when a thread is in the middle of a breakpoint. On a uniprocessor switched among several threads, the obvious interpretation is to prevent switching. On a uniprocessor running only a single thread, the interpretation is that the debugger must not use the thread's stack to call a procedure (e.g., during expression evaluation) while context switching is forbidden. On a shared-memory multiprocessor, only the processor running the thread that hit the breakpoint may be permitted to run; all the other processors must be stopped before the breakpointing thread executes instruction I.

The "debugger" in the model presented here is really monitoring, not debugging, because the breakpoint implementation always resumes execution immediately after encountering the breakpoint. A richer model would let the debugger decide when to resume execution, but it would not change the result. A real debugger must enable users to evaluate expressions when the target is stopped at a breakpoint. Such expressions include calls to procedures, which may themselves hit or "re-enter" the same breakpoint. To permit context switching from primary code to expression-evaluation code, the debugger must delay restoration of I until a suspended evaluation (or the original code) is ready to resume at I. When this delay is correctly implemented, a debugger can build up an arbitrary number of suspended evaluations, all of which have hit the breakpoint at I.

The PROMELA formalism and tools were designed to help validate network protocols, but they can be usefully employed on a wider range of problems. Designers and implementors of programming-environment tools should consider using PROMELA (or similar tools) to model interactions in their systems.

References

Aral, Ziya, Ilya Gertner, and Greg Schaffer. 1989 (May). Efficient debugging primitives for multiprocessors. *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems,* in a special issue of *SIGPLAN Notices*, 24:87–95.

Bruegge, Bernd. 1985 (September). *Adaptability and Portability of Symbolic Debuggers.* PhD thesis, Carnegie Mellon University.

Caswell, Deborah and David Black. 1990 (January). Implementing a Mach debugger for multithreaded applications. In *Proceedings of the Winter USENIX Conference*, pages 25–39, Washington, DC.

Digital Equipment Corporation. 1975. *DDT—Dynamic Debugging Technique.* Maynard, MA.

Dijsktra, Edsger W. 1976. *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall.

Gorlick, Michael M. 1991 (December). The flight recorder: An architectural aid for system monitoring. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging,* in *SIGPLAN Notices*, 26(12):175–183.

Hoare, C. A. R. 1978 (August). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

Holzmann, Gerard J. 1991. *Design and Validation of Computer Protocols.* Englewood Cliffs, NJ: Prentice Hall.

Kessler, Peter B. 1990 (June). Fast breakpoints: Design and implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 25(6):78–84.

Knuth, Donald E. 1984. Literate programming. *The Computer Journal*, 27(2):97–111.

Linton, Mark A. 1990 (June). The evolution of Dbx. In *Proceedings of the Summer USENIX Conference*, pages 211–220, Anaheim, CA.

Ramsey, Norman. 1992a (August). Literate-programming tools need not be complex. Technical Report CS-TR-351-91, Department of Computer Science, Princeton University. Submitted to *IEEE Software*.

———. 1992b (December). *A Retargetable Debugger.* PhD thesis, Princeton University, Department of Computer Science. Also Technical Report CS-TR-403-92.

Redell, David D. 1989 (January). Experience with Topaz TeleDebugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* in *SIGPLAN Notices*, 24(1):35–44.

Stallman, Richard M. and Roland H. Pesch. 1991. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA.

Wahbe, Robert. 1992 (September). Efficient data breakpoints. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* in *SIGPLAN Notices*, 27(9):200–212.