A Proof of *Incorrectness* using the LP Theorem Prover: The Editing Problem in the Therac-25

Muffy Thomas
Dept. of Computing Science
University of Glasgow
Glasgow, U.K.

August 11, 1993

Abstract

The Therac-25 is a computer-controlled radiation machine which delivered several accidental radiation overdoses, some of which resulted in the loss of life. As reported in [4], the source of the unsafe behaviour of the machine is a complex array of factors, one of which is an error in the set of editing routines for entering the treatment data. In this paper we formalise some of this code as a set of first-order axioms and show, using the automated theorem prover LP [1], that the code does not behave as intended, i.e. it is *incorrect*.

We do so by attempting to prove that the code does behave as intended, and then hope to learn about the nature of the error by examining how the proof fails. In order to gain confidence in our *lack* of proof, we begin by proving that some of the expected properties do indeed hold.

We conclude by using LP to guide us to a solution to the error and after correcting the code, we prove that the resulting code is indeed correct.

1 Introduction

Formal specification and verification techniques are part of a wider discipline of designing and implementing safety-critical software; by referring to a recent, relevant example, we hope to illustrate the contribution of formal methods. The example which we consider is the Therac-25: a computer-controlled radiation machine, or *linear accelerator*, used for radiation therapy. It was manufactured by Atomic Energy of Canada Ltd. (AECL) during the 1980s and was used at hospitals and clinics in the U.S.A. and Canada.

As a result of *software errors*, several patients were killed or injured by radiation overdoses delivered by the machine. The events leading up to these deaths and the actions taken as a result, are recounted in [3] and in a more detailed investigation of the accidents undertaken by N. Leveson and C. Turner in [4]. In another paper in this volume, [5], the machine is described in more detail.

Here, we consider only one small aspect of the Therac-25 software. It is not our intention to suggest that there are only a few isolated "bugs" in the Therac-25, or that formal specifications alone can capture all the potential behaviours and interactions of a safety-critical system; the sources of errors are usually very complex and influenced by a variety of factors, not all of which are quantifiable. However, we do feel that formal methods have an important role to play and the Therac-25 is a good application to study. No aspects of the Therac-25 machine have, to the author's knowledge, been formally specified (except in [5]).

Here, we consider one specific error, or "bug", as described in [4]. The problem concerns one particular aspect of the code for the machine: the editing of the treatment parameters. In [4], the relevant pseudocode (the actual code was written in assembler) is given. Our aim is to show, formally, using the automated theorem prover/proof assistant LP [1], that this code does not behave as intended, i.e. it is *incorrect*. We do so by attempting to prove that it *is* correct, and then hope to learn about the nature of the error by examining how the proof fails. In order to gain confidence in our proof, or lack of proof, we first ensure that we can prove that some of the properties that we expect to hold for the code do indeed hold.

The paper is organised as follows. In the next section, we give the background to the problem, the pseudocode for the relevant editing routines, and some correctness properties. Section 3 contains an overview of the LP theorem prover and in Section 4 we give the axiomatisation, in LP, of the editing routines. In Section 5 we prove some theorems about the correctness of the code and through the process of trying to prove further correctness conjectures, we are led to some theorems which illustrate how the code is incorrect.

These incorrectness results are discussed in Section 6 and then in Section 7 we use LP to direct our search for a correction to the error, concluding with a proof that the resulting code is correct. The results of our investigation are discussed in Section 8 and some conclusions are given in the final Section.

2 The Editing Problem

The Therac-25 delivers two kinds of radiation beams for radiation therapy: electron and X-ray. The electron beam is used to irradiate the patient directly, using scanning, or bending magnets to spread the beam to a safe and therapeutic concentration. The X-ray beam is created by bombarding a metal shield, or "beam flattener"; the electrons are absorbed by the shield and X-rays emerge from the other side. Since the efficiency of producing X-rays in this way is very poor, the current of the electron beam has to be increased to over 100 times the intensity when used directly for irradiation.

The greatest danger posed by the machine is the possibility of irradiating a patient directly with the high energy electron beam, i.e. without the X-ray shield in place. This situation could arise, for example, if both a high energy beam and an electron mode are specified as treatment parameters. Since such a combination is *usually* undesirable, we might expect the machine to recognise and block such a potentially dangerous situation. We now proceed to describe how this situation could, and did, arise in the Therac-25.

The machine operator enters the treatment data: treatment mode, energy level, position, etc., at the console before commencing treatment. If the operator wishes to alter the treatment data, for example, to rectify a mistake, then the data can be edited, using the cursor, without recommencing the data entry procedure from the beginning.

A problem uncovered in [4] can occur when the editing is carried out *very quickly*, i.e.whenthe operator moves the cursor, types in the new data, and returns the cursor, all within 8 seconds. It is possible, in this scenario, for the results of the edit to appear on the console screen but with the internal treatment parameters remaining unchanged.

In order to uncover exactly how this situation can arise, we will examine the high-level description of the various routines and tasks involved, as given in [4].

2.1 The Code

The overall behaviour of the machine is controlled by the TREAT task, which runs concurrently with the keyboard handler task. One of the first routines called by TREAT is DATENT (data entry) which communicates

```
{L1:}IF mode/energy in MEOS specified THEN
    BEGIN
        calculate table index
        REPEAT
            fetch parameter
            output parameter
            point to next parameter
        UNTIL all parameters set
        CALL MAGNET
    {L2:}IF mode/energy in MEOS changed THEN RETURN {goto L1}
    END
    IF data entry is complete THEN set TPHASE to 3
    IF data entry is not complete THEN
        IF reset command entered THEN set TPHASE to 0
    RETURN
```

Figure 1: DATENT ROUTINE

with the keyboard handler task via a shared variable to determine if the treatment data has been entered. When the data entry is complete, then the TREAT task begins another phase, namely to set up the machine and deliver the treatment. The variable TPHASE (treatment phas indicator) determines the next subroutine to be called: 0 for routine DATENT, and 3 for a routine called SET-UP TEST (a routine which is not relevant here).

The keyboard handler parses the mode and energy level specified by the operator and places the encoded result in the shared variable MEOS (mode/energy offset variable). Initially, the data entry process forces the operator to enter the mode and energy parameters, but the operator can later edit both of these parameters separately. If an edit to these parameters is detected during the rest of the data entry procedure, then the procedure should start over again.

When the energy and mode parameters are set, a table of preset operating parameters is consulted and fetched, according to the given mode and energy parameters. Once these operating parameters are set, the routine MAGNET is called which sets the bending magnets.

It takes a certain amount of time to set each magnet (about 8 seconds to

```
Set bending magnet flag
REPEAT
    Set next magnet
    Call PTIME
{L3:}IF mode/energy in MEOS has changed THEN exit {goto L2}
UNTIL all magnets are set
RETURN {goto L2}
```

Figure 2: MAGNET ROUTINE

```
      REPEAT
      IF bending magnet flag is set THEN
          IF editing taking place THEN
              IF mode/energy in MEOS has changed THEN exit {goto L4}
      UNTIL hysteresis delay has expired
{L4:}Clear bending magnet flag
      RETURN {goto L3}
```

Figure 3: PTIME ROUTINE

set all of them) and so the routine MAGNET calls another routine PTIME to introduce a time delay. A flag to indicate that magnet bending activity is taking place is set and cleared at the beginning and end (resp.) of PTIME. Since there are several magnets, PTIME is entered and exited several times. During both MAGNET and PTIME, the shared MEOS variable, which can be set by the keyboard handler, is consulted to determine whether or not the energy/mode parameters have been edited. There are numerous exit points, corresponding to detection of such edits.

The pseudocode for these routines: DATENT, MAGNET, and PTIME, is given in figures 1, 2 and 3, respectively. The code is taken from [4], with the some minor additions, for readability.

2.2 The Intended Behaviour

Our primary concern is that when edits to the energy/mode parameters occur, as reflected in the MEOS variable, they are detected within the DATENT routine and the routine should start again. That is, if the energy/mode parameter has changed, then control should return to $\{L1\}$. If no edits are detected, then after completing DATENT, control should return to TREAT, with a new TPHASE setting.

In essence, we are interested in two outcomes: return to the start of the DATENT routine again, or finish the DATENT routine. This outcome should depend on whether or not there are edits to the energy/mode parameters, as reflected in the MEOS variable, at *any* time during the execution of DATENT. More specifically, we are concerned with the edits which may occur during the magnet setting period. If the routines are *correct*, then

- if there are edits to the energy/mode parameters during the magnet setting period, then control should return to the start of the DATENT routine,
- if there are no edits to the energy/mode parameters during the magnet setting period, then the DATENT routine should finish and return control to TREAT.

In the next section, we give a formalisation of the code which will allow us to check, formally, whether or not the code behaves correctly, with respect to these properties.

3 The LP Theorem Prover

LP [1] (the *Larch Prover*) is an interactive proof assistant for a subset of multisorted first-order logic. The interested reader is directed to [2] for a more comprehensive introduction to LP. Here, we review how LP was used in this case study.

The underlying language is defined in LP by *declarations*. Sorts, variables, and operators (with functionality) are declared by commands of the form:

```
declare sort Elem
declare variables x,y: Elem
declare operators
  c: -> Elem
  f: Elem -> Elem
  ..
```

Multi-line commands in LP are always terminated by "..".

Axioms are defined in LP by *assertions*. In our formalisation, we use three forms of axiom: equations, which are then oriented as rewrite rules, rewrite rules, and induction rules. Axioms are defined in LP by commands of the form:

```
assert
  f(c) == c
  f(c) -> c
  ..
```

for equations and rewrite rules, respectively, and by:

```
assert
  Elem generated by c,f
```

for induction rules.

Much of LP's deductive system is based on the rewrite rules which result from orienting the equations. In general, not all equations are orientable; however, in this example they may be oriented using the built-in (simplification) termination ordering *noeq-dsmpos*. This is a registered ordering based on the partial ordering between operators given by the user, and the one deduced by LP according to the rewrite rules seen so far. In our axiomatisation, we give equations when we are certain that there is enough information in the registry to deduce the orientation we intend. This is the preferred approach. Otherwise, we give the axiom as a rewrite rule.

There are numerous inference methods in LP and several methods of *backward* inference. In this example, the proofs of theorems are carried out using a combination of **prove by case** and **prove by implication** methods, along with rewrite rule **instantiation** and **normalisation**.

4 The Axiomatisation

There are two obvious ways in which to formalise the editing code: formalise an abstract machine for the code and then consider the behaviour of this machine when "running" the code, or formalise the code directly, as an operation on abstract machine states. Since the editing code is the only example program to be considered, we choose the latter approach.

The main components of the formalisation are (machine) *states* and *editing histories*. The former reflect the state of the (abstract) machine; thus, the routines are operations on states. The latter reflect the behaviour of the operator at the console. Essentially, we must model the *editing* behaviour, at the console, over the time taken to set the magnets. Namely, for each magnet, for each moment of the delay required for setting that magnet, we need to specify whether or not the energy/mode parameters have been edited.

The remaining two components of the formalisation are *numbers* and the *magnets* and *delay*, which we model using numbers. Without loss of generality, we fix the number of magnets to 3 and the unit of time delay to 1.

In the formalisation, we have abstracted away from the details of the *actual* data concerned; eg. the values of the mode and energy parameters and other parameters which may be set by the operator, and details concerning how the magnets are bent. Rather, we are only concerned with whether or not the values have changed. Thus, the relevant parameters are modelled by boolean variables.

In the following subsections, we give the LP axiomatisations of machine states, numbers, magnets and delay, editing histories, and finally, the code.

Comments are given on a line beginning with `%` and the current name used to prefix rules and conjectures is defined by the command `set name` *name*.

4.1 State

For our purposes, an abstract state (the sort `state`) of the machine need contain only three pieces of information: whether or not the mode/energy parameters in the MEOS variable have changed, whether or not the state of the magnet bending flag is set, and whether or not editing is taking place. These are each represented by variables of the (built-in) boolean type and named MEOS, MF (for magnet flag) , and EF (for editing flag), respectively. MEOS is true only when the mode/energy parameters have changed; MF is true only when the magnet bending flag is set; and EF is true only when editing is taking place (that is, editing to any data field and not just to the MEOS variable).

There is one generator operation to make a `state`; in addition, there are operations to set and clear the MEOS and MF components of a state, and several predicates to check the states of the components MEOS, MF and EF. Note that there is no operation to set the edit flag, EF, since this is not a program operation but an operation of the keyboard handler.

```
set name basicstate
declare sort state
declare variables MEOS,MF,EF: bool, st,st1,st2: state

declare operators
  % a state is a triple: <MEOS, MF, EF>
  % MEOS- mode/energy offset variable (true when editing mode or energy)
  % MF  - magnet flag  (true when bending magnets)
  % EF  - edit flag    (true when editing any data entry)

  stat: bool, bool, bool -> state
  setMF : state -> state
  setMEOS : state -> state
  clrMF : state -> state
  clrMEOS : state -> state
  issetMF : state -> bool
  issetMEOS : state -> bool
  issetEF : state -> bool
  issetEFMEOS : state -> bool
  ..
assert state generated by stat
```

```
assert
  setMF(stat(MEOS,MF,EF))  == stat(MEOS,true,EF)
  setMEOS(stat(MEOS,MF,EF)) == stat(true,MF,EF)
  clrMF(stat(MEOS,MF,EF)) == stat(MEOS,false,EF)
  clrMEOS(stat(MEOS,MF,EF)) == stat(false,MF,EF)
  issetMF(stat(MEOS,MF,EF)) = MF
  issetMEOS(stat(MEOS,MF,EF)) == MEOS
  issetEF(stat(MEOS,MF,EF)) == EF
  issetEFMEOS(st) == issetEF(st) & issetMEOS(st)
  ..
```

4.2 Numbers

The formalisation of numbers is fairly uninteresting. We use the standard LP file "nat.lp" and add two comparison operators and the constant "3". The interested reader can find the script for nat.lp in the Appendix.

```
execute nat.lp

declare operators
  > :nat, nat -> bool
  <=: nat, nat -> bool
  3 :-> nat
 ..
assert
  s(x) > 0 -> true
  0 > s(x) -> false
  0 > 0 -> false
  s(x) > s(y) -> x > y
  x <= y -> y > x  | x = y
  3 -> s(2)
```

4.3 Magnets and Delay

As we stated above, magnets and delays are modelled by numbers.

```
set name magnets
declare operators
  magnets: -> nat
assert
```

```
  magnets == 3

set name delay
declare operators
  delay :-> nat
assert
  delay == 1
```

4.4 Editing Histories

An editing history (the sort `edithistory`) models the (relevant) history of the editing behaviour at the console, during the setting of the magnets.

An editing state, at any moment, is represented by the sort `editpr`, pairs of booleans representing whether or not the mode/energy parameters have been changed, as reflected in the variable MEOS, and whether or not an editing is taking place. These components are called MEOS and EF, as before. A state in which the mode/energy parameters are being edited is represented by both MEOS and EF being true; if only EF is true, then some other part of the treatment data is being edited.

An editing history, sort `edithistory`, is essentially a list of `editpr`: the positions in a list determining the corresponding magnet and delay. Since, in this case, the delay is 1, the positions in the list correspond directly to the magnet numbers. Moreover, since we are restricted to lists of length 3, we can model editing histories by triples. Thus, there in one generator operation to make an `editpr`, `pr`, and one generator operation to make an `edithistory`, `edit`. There is one selector operation `select` which returns an edit pair, from an edit history, given a magnet number and delay. Since the delay is 1 in this example, an auxiliary operation `selectx` is used to define `select`. The constants `edit1`, `edit2`, and `edit3` are used for example edit histories. The particular examples are described in the comments in the axioms where they defined.

There are four predicates on edit histories: `iseditmeos` and `noeditmeos` denoting (resp.) whether or not there is an edit to the energy/mode parameters at any time in the history, and `mag3editmeos` and `nomag3editmeos` denoting (resp.) whether or not there is an edit to the energy/mode parameters during the setting of magnet 3. The motivation for these last two predicates will be revealed when we try to prove the correctness properties.

Finally, there are two predicates on edit pairs: `issetMEOS`, and `issetEF`, which are similar to the predicates on `state` with the same name.

```
set name editing
declare sort edithistory, editpr
declare operators
    %edit - make an edit history
    %edit1, edit2, edit3 - constants for example edits
    %select: magnet,delay, edithistory -> state
    %iseditmeos - any edit to mode/energy?
    %noeditmeos - no edit to mode/energy?
    %mag3editmeos - an edit to mode/energy during magnet 3?
    %nomag3editmeos - no edit to mode/energy during magnet 3?
    %issetMEOS,issetEF, issetEFMEOS - edits to EF and/or MEOS?

    edit : editpr, editpr, editpr -> edithistory
    pr : bool, bool -> editpr
    select : nat, nat, edithistory -> editpr
    selectx : nat, edithistory -> editpr

    iseditmeos, noeditmeos : edithistory -> bool
    nomag3editmeos : edithistory -> bool
    mag3editmeos : edithistory -> bool
    issetMEOS : editpr -> bool
    issetEF : editpr -> bool
    issetEFMEOS : editpr -> bool
  ..
%ordering information
register height  (edit3,edit2, edit1) > edit
register height edit > (stat,true,false,1,2)
register height noeditmeos > iseditmeos
declare variables m,n : nat, est : edithistory, e1,e2,e3 : editpr
declare variables MEOS1,MEOS2, MEOS3:bool, MF1,MF2,MF3 : bool
declare variables EF1,EF2,EF3 : bool
assert edithistory generated by edit
assert
    select(s(x),y,est) == selectx(s(x),est)  %assume that y=1
    selectx(1,edit(e1,e2,e3)) == e1
    selectx(2,edit(e1,e2,e3)) == e2
    selectx(3,edit(e1,e2,e3)) == e3
    iseditmeos(edit(e1,e2,e3))
          == issetEFMEOS(e1) | issetEFMEOS(e2) | issetEFMEOS(e3)

    noeditmeos(est) == not(iseditmeos(est))

    nomag3editmeos(edit(e1,e2,e3)) == not(issetEFMEOS(e3))
    mag3editmeos(edit(e1,e2,e3)) == issetEFMEOS(e3)

    issetMEOS(pr(MEOS,EF)) == MEOS
    issetEF(pr(MEOS,EF)) == EF
    issetEFMEOS(e1) == issetMEOS(e1) & issetEF(e1)
  ..

% define some editing behaviours as a constants
% define edits in order, eg, edit(1st pr,2nd pr,3rd pr)

assert
    %edit1:  3  magnets, 1 delay,  no edits
    edit1 == edit(pr(false,false),pr(false,false),pr(false,false))

    %edit2: 3 magnets, 1 delay, edit to MEOS during magnet 3
    edit2 == edit(pr(false,false),pr(false,false),pr(true,true))

    %edit3: 2 magnets, 1 delay, edit to MEOS during magnet 1 and
    % magnet 2, but no edit during magnet 3
     edit3 == edit(pr(true,true),pr(true,true),pr(false,false))
 ..
```

4.5 Code

Now we are able to formalise the code. The operations are essentially operations from `state` to `state`, with the additional argument of a current edit history, and, in some cases, loop counters and indices for selection on the edit history. The operations correspond quite closely to the program statements and routines, with two exceptions. The first is that in order to detect a return to the start of DATENT, i.e. to $\{L1\}$, we introduce a constant state `STARTAGAIN` and replace the RETURN in $\{L2\}$, i.e. *goto* $\{L1\}$, by the state

STARTAGAIN. The second is that in order to represent a return of control to TREAT, after completing DATENT, we introduce a state FINISH(_). FINISH(_) depends on the current state, so that we can observe the state in which the program finishes. We replace the commands after the main loop of DATENT, by FINISH(st), and so the statement {*L2*} involves a branching between STARTAGAIN and FINISH(st). We explicitly assert that STARTAGAIN and FINISH(st) are not equivalent.

We note that the magnets are processed in the order magnet down to 1; i.e. in this example, magnet 3 is processed first.

Although we have included the operations Fetchparams and Tableindex for completeness, these operations are not relevant here and they have no observable effect on the state.

```
set name code

declare operators
  L1 : edthistory ->state
  L2 : state -> state
  L3 : edthistory, nat, state -> state   % nat is for loop counter
  L4 : edthistory, nat, state -> state   %  ""

% subroutines
  MAGNET : edthistory, state -> state
  PTIME : edthistory, nat, nat, state -> state % nats for loop counters

% others
  MAGLOOP : edthistory, nat, state -> state   % nat for loop counter
  BExp1 : edthistory, nat, nat,state -> state % nats indices for history
  BExp2 : edthistory, nat, nat,state -> state %  ""
  Fetchparams :  state -> state
  Tableindex  :  state -> state

% state constants
  INITIAL :-> state
  STARTAGAIN :-> state
  FINISH : state -> state

assert
INITIAL -> stat(false,false,false)
```

```
L1(est) -> MAGNET(est,Fetchparams(Tableindex(INITIAL)))

Fetchparams(st) -> st
Tableindex(st) -> st

MAGNET(est,st) ->MAGLOOP(est,magnets,setMF(st))

MAGLOOP(est,0,st) == L2(st)
MAGLOOP(est,s(x),st) == PTIME(est,s(x),delay,st)

PTIME(est,x,0,st) -> L4(est,x,st)
PTIME(est,x,s(y),st)
      -> if(issetMF(st),BExp1(est,x,s(y),st),PTIME(est,x,y,st))
BExp1(est,x,s(y),st)
     -> if(issetEF(select(x,s(y),est)), BExp2(est,x,s(y),st),
             PTIME(est,x,y,st))
BExp2(est,x,s(y),st)
     -> if(issetMEOS(select(x,s(y),est)), L4(est,x,setMEOS(st)),
            PTIME(est,x,y,st))

L2(st)-> if(issetMEOS(st), STARTAGAIN, FINISH(st))

L3(est,s(x),st) -> if(issetMEOS(st), L2(st), MAGLOOP(est,x,st))

L4(est,x,st)-> L3(est,x,clrMF(st))
```

4.6 Ordering Register

The partial ordering on operators is given by:

```
register height magnets > (2,s)
register height delay > 1
register height (edit3,edit2, edit1,edit) > (stat,true,false,1,2)
register height noeditmeos > iseditmeos
register height BExp1 > PTIME
register height PTIME > L3
register height  L4 > L3 > (STARTAGAIN, MAGLOOP)
register height MAGLOOP > L2
register height L1 > FINISH
```

Given this register information, the equations and rewrite rules can be ordered, as desired, according to the termination ordering. When the equations are oriented as rewrite rules, and internormalised, some of the rules are very large indeed, eg. they can only be displayed over several screens on the terminal. It is quite clear that it is not feasible to carry out proofs in this rewriting system by hand; some machine assistance is obligatory.

5 Theorems and Proofs

Now consider some of the desirable properties of the code. The properties which interest us are of the form "if there are no edits (to the mode/energy parameters in the MEOS) during the magnet setting period then the program finishes".

In our formalisation, we represent an edit by the open formula `edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3))`, where `MEOS1`, `EF1` ... `MEOS3`, `EF3` are boolean variables representing the variables of the same name during the setting of each magnet. We represent "there are no edits", for a given edit history, by the (open) formula `noeditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))`.

Since the program depends on an edit history and begins with the statement $\{L1\}$, the program is represented by the (open) formula `L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))`. This is the general form of the program with an edit history.

The program with an example history, eg. with example edits `edit1,` `edit,` or tt edit3, is given more simply by a formula of the form `L1(edit1)`.

In the following subsections, we give some example conjectures and proofs. Some proofs require no user assistance, whereas others require guidance. In each case, the guidance is fairly minimal. The descriptions which follow are sequences of LP commands, with only minimal discussion of how the proofs are carried out. The reader is again directed to [2] for a detailed introduction to theorem proving in LP.

The `resume by case` x command causes LP to resume the proof (of the current (sub) goal) under the assumptions of the case hypotheses. Namely, resume the proof first under assumption x true, and then under assumption not x true. The `qed` command causes LP to check whether there are any outstanding conjectures; if so, then an error message appears. If not, there is no response (except to expect the next command) and we may interpret this as a "mathematical" *qed*.

In the proofs, the lines beginning with `[]` (box) and `<>` (diamond) are LP annotations. Diamonds indicate the introduction of subgoals and boxes indicate the discharging of subgoals. When replaying a proof, these annotations behave like the `qed` command: they cause an error to be reported if the proof does not follow the indicated format.

5.1 Correctness Properties

We begin by checking that some simple properties hold; for example, we prove that `edit3` contains some edits to MEOS, but they do not take place during the setting of magnet 3. These properties are given in LP by the following conjectures, or theorems, $C1$ and $C2$, which are proved by normalisation:

$\underline{C1:}$

```
prove nomag3editmeos(edit3) = true
  [] conjecture
qed
```

$\underline{C2:}$

```
prove iseditmeos(edit3) = true
  [] conjecture
qed
```

We can also prove that there are no edits to MEOS in edit history `edit1`, and that the program, with `edit1`, finishes in a state of the form `FINISH(st)`. In fact, it finishes in a state with all three variables false. This is given in LP by the theorems:

$\underline{C3:}$

```
prove noeditmeos(edit1) = true
  [] conjecture
qed
```

$\underline{C4:}$

```
prove L1(edit1)  = FINISH(stat(false,false,false))
  [] conjecture
qed
```

Now we try to generalise theorem $C4$ to the first of the correctness properties mentioned above: if there are no edits in the edit history then the program finishes, i.e. it ends in a state of form FINISH(stat(false,false,false)). This is given in LP by the implication:

$\underline{C5:}$

```
prove
  (noeditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3))))
     =>
     L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
         = FINISH(stat(false,false,false))
     by =>
  ..
```

We discover that we cannot prove this conjecture directly from the axioms, we must first prove the following two lemmata. The first is proved by implication and normalisation and the second is proved by case and normalisation:

$\underline{C6:}$

```
prove
  noeditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    =>
    not(MEOS3 & EF3)
    by =>
  ..
  <> 1 subgoal for proof of =>
     [] => subgoal
  [] conjecture
qed
```

$\underline{C7:}$

```
prove not(b1 & b2) => if(b1,if(b2,st1,st2),st2) = st2
  resume by case b2
  <> 2 subgoals for proof by cases
    resume by case b1
    <> 2 subgoals for proof by cases
      [] case b1c
      [] case not (b1c)
    [] case b2c
    [] case not(b2c)
  [] conjecture
qed
```

Now the main conjecture, $C5$, is proved by instantiation, referring to lemma $C7$ which is named `code.17` in LP. This instantiation is necessary as during the course of proving $C5$, its variables are replaced by constants.

```
prove
  (noeditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3))))
     =>
     L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
         = FINISH(stat(false,false,false))
     by =>
  ..
  <> 1 subgoal for proof of =>
    instantiate b2 by MEOS3c,b1 by EF3c in code.17
    [] => subgoal
  [] conjecture
qed
```

Another property to check is that when there is an edit to the energy/mode parameters in MEOS, as in edit2, then the program starts again, i.e. it ends in a state of form `STARTAGAIN`. These properties are given in LP by the following theorems which are all proved by normalisation:

<u>$C8$:</u>

```
prove iseditmeos(edit2) = true
  [] conjecture
qed
```

<u>$C9$:</u>

```
prove mag3editmeos(edit2) = true
```

```
  [] conjecture
qed
```

<u>$C10$:</u>

```
prove L1(edit2) = STARTAGAIN
  [] conjecture
qed
```

Now consider generalising this last theorem to *any* edit history. As a first step in generalisation, consider the case where there is an edit during magnet 3; i.e. if there is an edit to MEOS, then the program starts. This is given in LP by:

<u>$C11$:</u>

```
prove
  mag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    =>
    L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
      = STARTAGAIN
  ..
```

Again, this conjecture does not follow from the axioms, but when we first prove the subgoals:

<u>$C12$:</u>

```
prove
  mag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
     =>
    (MEOS3 & EF3)
   by =>
  ..
  <> 1 subgoal for proof of =>
    [] => subgoal
  [] conjecture
qed
```

<u>$C13$:</u>

```
prove
  mag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    =>
    L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
      = STARTAGAIN
  ..
  <> 1 subgoal for proof of =>
    [] => subgoal
  [] conjecture
qed
```

then theorem C11 is proved by:

```
prove
  mag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    =>
    L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
      = STARTAGAIN
  ..
  <> 1 subgoal for proof of =>
    instantiate b2 by MEOS3c,b1 by EF3c in code.17
    [] => subgoal
  [] conjecture
qed
```

We can generalise this theorem further to the main conjecture for the correctness property, i.e. if there are *any* edits to MEOS then the program starts again by:

$\underline{C14:}$

```
prove
  (iseditmeos(edit(b1, b2, b3, b4, b5, b6, b7, b8, b9))
     =>
     L1(edit(b1, b2, b3, b4, b5, b6, b7, b8, b9))
       = STARTAGAIN
  ..
```

We find that we are unable to prove this theorem; and from the subgoals generated by LP, we are led to think of the following conjecture:

$\underline{C15:}$

```
prove
  (iseditmeos(edit(b1, b2, b3, b4, b5, b6, b7, b8, b9))
  & nomag3editmeos(edit(b1, b2, b3, b4, b5, b6, b7, b8, b9)))
     =>
     L1(edit(b1, b2, b3, b4, b5, b6, b7, b8, b9))
       = STARTAGAIN
  ..
```

This conjecture says that *if* there is an edit to the MEOS, but it is *not* during the setting of magnet 3, then the program starts again. This is just a refinement of $C14$ and so it is also a *correctness* property that should hold. But, again, we *cannot* prove it.

5.2 Incorrect properties

We cannot prove the main correctness property $C14$, nor the conjecture $C15$, because we *can* prove that when there is an edit to the MEOS, but *not* during the setting of magnet 3, then the program *finishes*. Surely, this is not an intended behaviour as we wish the program to start again *whenever* the MEOS has been edited. In LP, we prove it as follows:

$\underline{C16:}$

```
prove
  (iseditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
   &
  nomag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3))))
      =>
     L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
                 = FINISH(stat(false,false,false))
  ..
  resume by case MEOS3 & EF3
  <> 2 subgoals for proof by cases
    [] case EF3c & MEOS3c
    instantiate b2 by MEOS3c, b1 by EF3c in code.17
    [] case not(EF3c &MEOS3c)
  [] conjecture
qed
```

A hint of what is going wrong in the program comes when we show the normal form of the term

`L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))`,
the general form of the program with an edit history.

$\underline{C17:}$

```
show normal
  L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
```

gives the term

```
    if(EF3,
       if(MEOS3, STARTAGAIN, FINISH(stat(false, false, false))),
       FINISH(stat(false, false, false)))
```

This normalisation demonstrates that the behaviour of the program depends *only* on the variables `MEOS3` and `EF3`: the variables used to define the state during the setting of magnet 3. The other variables representing the edit history during the setting of the other magnets are ignored. This means that the behaviour at the console, during the setting of the other magnets, is *ignored*! Thus, for example, an edit to the MEOS during the setting of magnet 2 will be ignored and the program could finish, instead of starting again.

Indeed, this result leads us to try to prove the conjecture which says that if there are no edits during the setting of magnet 3, then the program finishes. In LP this is proved in two steps by:

$\underline{C18:}$

```
prove
  nomag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    => not(MEOS3 & EF3)
  by =>
  ..
  <> 1 subgoal for proof of =>
    [] => subgoal
  [] conjecture
qed
```

$\underline{C19:}$

```
prove
  nomag3editmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    => L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
              = FINISH(stat(false,false,false))
  ..
 [] conjecture
qed
```

We conclude from these theorems that the behaviour of the routines depends *solely* on the editing behaviour at the console during the setting of magnet 3, i.e. the first magnet to be set.

6 Discussion of Incorrectness

Since the behaviour of the editing routines depends solely on the editing behaviour during the setting of the first magnet (i.e. magnet 3), if an edit occurs during the setting of the other magnets, then the edit will be ignored in routine DATENT. In practice, since setting the magnets takes about 8 seconds (the precise time is not known by this author), this means that edits performed within 8 seconds of initial data entry, but after the time to set one magnet, although appearing on the console, are not reflected in DATENT by the fetching of the relevant operating parameters. Thus, without hardware interlocks (which were present in the predecessor Therac-20 machine but not in the Therac-25), a radiation overdose is, and was, possible.

7 Corrections

We can pinpoint the problem in the program to the clearing of the bending magnet flag at $\{L4\}$ in PTIME. As discussed in [4], this flag is cleared after the first magnet is set and so in futher calls of PTIME, the premise of the first conditional: IF bending magnet flag is set THEN ..., will always be false. Thus, edits during the remaining calls of PTIME cannot detected.

In this section we use LP to try to pinpoint the source of error in order to correct it. We have shown, above, that the behaviour depends solely on the editing state during the setting of the first magnet. In order to uncover the error, we will work our way backwards through the behaviour and examine the state of the machine at two points: before the first call of PTIME from MAGNET and after one call of PTIME from MAGNET.

The first examination point is revealed by replacing the rule

```
MAGLOOP(est,s(x),st) == PTIME(est,s(x),delay,st)
```

by

```
MAGLOOP(est,s(x),st) -> st
```

and then showing the normal form of

```
L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
```

which gives:

```
stat(false,true,false).
```

This shows us that before PTIME is called (for the first time), regardless of the editing history, the MF flag (the second parameter to `stat`) is `true`. This is as we expect: the magnet bending flag *should* be true during the setting of the magnets.

Now consider the second examination point: after the first call of PTIME. This point is revealed by replacing the rule

```
L3(est,s(x),st) -> if(issetMEOS(st), L2(st), MAGLOOP(est,x,st))
```

by

```
L3(est,s(x),st) -> st
```

(the remaining rules are in their original form) and then showing the normal form of

```
L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
```

which gives:

```
if(EF3,
  if(MEOS3,stat(true,false,false),stat(false,false,false)),
  stat(false,false,false)).
```

This shows us that in each possible outcome, regardless of the values of `EF3` and `MEOS3`, the MF flag (the second parameter to `stat`) is `false`. However, our intuition tells us that the magnet bending flag should only be cleared *after* the bending of the magnets has finished. Thus, we conclude that the magnet bending flag has been incorrectly cleared after one call of PTIME.

Our proposed correction is to clear the flag within the MAGNET routine. Thus, we replace the rule which clears the magnet bending flag:

```
L4(est,s,st) -> if(est,x,clrMF(st))
```

by

```
L3(est,x,st) -> if(est,x,st)
```

and replace the rule which calls L2 at the end of MAGNET with the cleared magnet bending flag:

```
MAGLOOP(est,0,st) == L2(clrMF(st))
```

by

```
MAGLOOP(est,0,st) == st.
```

7.1 Proving the Corrections Correct

We now show that this code *is* correct by proving that the two correctness properties hold; i.e. we show that conjectures C5 (if there are edits in the edit history then the program finishes) and C14 (if there are edits in the edit history then the program starts again) hold for the modified code.

In order to prove these conjectures, the following lemma is required.

$\underline{C20:}$

```
prove (b1 & b2) => if(b1,if(b2,st1,st2),st3) = st1
  resume by case b1
  <> 2 subgoals for proof by cases
    resume by case b2
    <> 2 subgoals for proof by cases
      [] case b2c
      [] case not(b2c)
    [] case b1c
    [] case not(b1c)
  [] conjecture
```

```
qed
```

The proofs of the conjectures (renamed C21 and C22, respectively) are carried out using case analysis and proof by implication.

<u>$C21$:</u>

```
prove
  (iseditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3))))
  =>
  L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
    = STARTAGAIN
  ..
  resume by case EF3 & MEOS3
  <> 2 subgoals for proof by cases
    [] case EF3c & MEOS3c
    resume by case EF2 & MEOS2
    <> 2 subgoals for proof by cases
      [] case EF2c & MEOS2c
      resume by case EF1 & MEOS1
      <> 2 subgoals for proof by cases
        [] case EF1c & MEOS1c
        [] case not(EF1c & MEOS1c)
      [] case not(EF2c & MEOS2c)
    [] case not(EF3c & MEOS3c)
  [] conjecture
qed
```

<u>$C22$:</u>

```
prove
  noeditmeos(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3)))
   =>
   L1(edit(pr(MEOS1,EF1),pr(MEOS2,EF2),pr(MEOS3,EF3))) =
  FINISH(stat(false,false,false)) by =>
  ..
  <> 1 subgoal for proof of =>
    instantiate b1 by EF3c, b2 by MEOS3c in code.18
    instantiate b1 by EF2c, b2 by MEOS2c in code.18
    instantiate b1 by EF1c, b2 by MEOS1c in code.18
    [] => subgoal
  [] conjecture
qed
```

8 Discussion

We believe that this example illustrates one of the best applications of automated theorem proving: to uncover an error in a piece of software (or hardware). Anecdotal evidence in support of this approach is that we did try informally, along with a few colleagues, to locate the error before carrying out the formal exercise, but without success. We note that it is reported in [4] that the error was fixed in a similar way, *after* the accidents, by clearing the MF flag after all the magnets have been set, in the routine MAGNET.

However, we must be clear about what we have and have not proved. Correctness is a relative property and we have only discovered the error because we were considering a particular correctness property. Thus, verification should be seen as the search for evidence that certain incorrect (or indeed unsafe) situations do no arise, rather than a search for the unobtainable - a proof of absolute correctness, or safety (the role of formal methods in reasoning about complex systems is further discussed in [6]). Unfortunately, poor software engineering methods were used in the development of the Therac-25 code, and certainly no formal specification, or verification, was carried out.

Moreover, we have to be very careful when proofs fail: do they fail because the conjectures are *really* untrue, w.r.t. the real life object, or do they fail because there is insufficient theory?

In this example, we were confident of our result of incorrectness, with respect to the original code, for two reasons:

- we had enough theory to prove some of the expected properties,
- we were able to prove another result, C19 (the theorem which says that if there are no edits during the setting of magnet 3, then the program finishes), which shows why the proof of the correctness property C15 fails.

The experience with LP was a very positive one, and we have a high degree of confidence in our results because there were achieved using this theorem prover/checker. The axiomatisation described herein is *not* our first attempt. In a previous axiomatisation, we used a different, more abstract representation of editing histories. Whilst this representation was more appealing, as a specification, the work required to build up enough theory to enable the proofs of the basic correctness properties was very discouraging. As a result, that approach was abandoned and the current approach using histories as lists was adopted. Once this more operational style of specification was used, the theorem proving tasks were quite straightforward, given a moderate knowledge of how to use LP.

9 Conclusions

We have formalised a pseudocode description of some (assembler) code known to be a source of error in the computer-controlled Therac-25 radiation machine, and used the theorem prover/proof assistant LP to reason about the behaviour of the code. Our strategy was to try to prove enough of the expected properties of the code so that we could learn about the nature of the error when a proof of a desired, but untrue, property failed. The strategy was successful and we were able to uncover exactly why the code did not behave as expected. We then went on to correct the error and prove that the corrected code was indeed correct.

This approach can only be considered successful when we a) consider a useful set of desired properties and b) have confidence that the theory we consider is rich enough with respect to the real life object. In this example, we have demonstrated both these aspects.

The axiomatisation and proofs involved in the example could not be managed by hand: the theorem prover LP was a very valuable and reliable tool.

Finally, this application of formal methods is just one part of an approach to dealing with safety-critical software and hardware systems. We agree with Leveson and Turner that with respect to safety-critical systems, we should not focus too much on a particular software error alone; we must also consider the whole complex system of which the software is just one component.

Acknowledgements

This work was carried out whilst the author was visiting Digital Equipment Corporation Systems Research Centre (DEC SRC), Palo Alto, California. Jim Horning, Jim Saxe and Steve Garland provided valuable assistance in the use of LP. Financial support from DEC SRC and The Nuffield Foundation is gratefully acknowledged.

References

[1] S. Garland, J. Guttag, An Overview of LP, the Larch Prover, Proceedings of the Third International Conference on Rewriting Techniques and Applications, Chapel Hill, N.C., Lecture Notes in Computer Science 355, Springer-Verlag, pp.137-151, 1989.

[2] S. Garland, J. Guttag, A Guide to LP, The Larch Prover, Report no. 82, Digital Equipment Corporation Systems Research Center Research Reports, 130 Lytton Avenue, Palo Alto, California, 1991.

[3] J. Jacky, Safety-Critical computing: Hazards, Practices, Standards and Regulation, in Computerization and Controversy, Dunlop and Kling (eds.), Academic Press, 1991.

[4] N. Leveson, C. Turner, An Investigation of the Therac-25 Accidents, IEEE COMPUTER, pp. 18-41, July 1993.

[5] M. Thomas, The Story of the Therac-25 in LOTOS, High Integrity Systems Journal, Volume 1, Issue 1, OUP, 1993.

[6] M. Thomas, Order, Disorder and Chaos in Complex Systems: The Role of Formal Methods in Safety-Critical Computer Software, Computing Science Research Report, Formal Methods Sub-Series, FM-1993-6, University of Glasgow, 1993.

A Appendix

```
% Axioms for the natural numbers in nat.lp

set name nat

declare sort Nat
declare variables x, y, z: Nat
declare operators
  0, 1, 2 :           -> Nat
  s       : Nat       -> Nat
  +, *    : Nat, Nat -> Nat
  ..

% Ordering hints

register height * > + > 2 > 1 > s > 0
register polynomial 0     2             2
register polynomial 1     5             5
register polynomial 2     8             8
register polynomial +     x + y + 1     x*y
register polynomial s     x + 2         x + 2
register polynomial *     x*y           x*y

% Axioms

assert ac +
assert ac *
assert Nat generated by 0, s

assert
  x + 0 == x
  x + s(y) == s(x + y)
  x * 0 == 0
  x * s(y) == (x * y) + x
  x * (y + z) == (x * y) + (x * z)
  1 == s(0)
  2 == s(1)
  0 = s(x) == false
  s(x) = s(y) == x = y
  ..
```