The Story of the Therac-25 in LOTOS

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

August 12, 1993

Abstract

We consider the use of formal specification and verification techniques for proving the safety, or otherwise, of an abstraction of a safety-critical medical application: the Therac-25 radiation machine. This machine was responsible for several patient deaths in the late 1980s. The specification is given in LOTOS and we consider trace analysis, property testing, and temporal logic for reasoning about the safe and unsafe behaviour of the specified machine. The testing tool LOLA is used for rigorous verification; with LOLA, two significant design errors are uncovered. The work reported herein is part of a case study on the practical use of formal methods in safety-critical software; the specification is based only on an informal description of part of the machine's behaviour, and does not constitute a specification of the entire machine.

1 Introduction

The Therac-25 is 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 a paper by J. Jacky [7] and more recently, a detailed technical account of the accidents is given by N. Leveson and C. Turner in [8].

In this paper, we will use the Therac-25 as the basis of a case-study in the formal specification and verification of a safety-critical medical application. The specification language is LOTOS (*Language of Temporal Ordering Specification*) [6]. LOTOS is an ISO standardised specification language which allows for both the specification of concurrent, nondeterministic processes and algebraic data types.

The actual source(s) of errors in the Therac-25 is a complex issue, as shown in [8]. In another paper in this volume (see [12]) we have taken one particular aspect of the code for the machine, namely the editing of the treatment parameters, and have used an automated theorem prover to show that it does not behave as intended. It is not our intention here to suggest that there are only a few isolated 'bugs' in the Therac-25, or that one formal specification alone can capture all the potential behaviours and interactions. However, formal specification and verification techniques are part of a wider discipline of designing and implementing safety-critical software, and by referring to a recent, relevant example, we hope to illustrate the contribution of formal methods. To the author's knowledge, no aspect of the Therac-25 machine has been formally specified before.

The overall aim of this paper is to specify the high-level behaviour of the Therac-25, complete with design errors, and to try to use formal methods to uncover these errors. More specifically, our aims are to

- develop a specification of the behaviour of the Therac-25,
- formalise some safety properties for the specification,
- attempt to formally prove/disprove that these safety properties hold,
- use the specification and verification to 'tell the story' of the Therac-25 tragedy.

The paper is organised as follows. In the next section, we give an informal description of the Therac-25 and an overview of the LOTOS specification language. We briefly present the syntax and semantics of the language, and the notion of testing in LOTOS. In Section 3 we give the first formal specification of the machine; we formalise the safety property and prove, using testing, that the specification permits unsafe behaviour. In Section 4 we develop another, safer, specification and discuss the use of testing for verification. In Section 5 we add interaction with the user and show how, combined with the specification of the previous section, the specification becomes unsafe.

In Section 6 we consider how testing and temporal logic might be used to verify the safety properties. In the final section we discuss our results and plans for future work.

2 Background

2.1 Therac-25

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 intensity electron beam, i.e. without the X-ray shield in place. This is a well-known danger, particularly since 1966 when a control failed on a traditional, electromechanical radiation machine at Hammersmith Hospital [2], and several patients were overdosed in this way. The Therac-25 differs from its predecessor the Therac-20 in an important aspect. Whereas the Therac-20 is also computer-controlled, it still has an independent set of electromechanical interlocks for ensuring safe operation; in the Therac-25, all monitoring is carried out by the software.

Of course another danger could be posed by *underdosing*, as then the underlying disease would not be treated as prescribed. However, we will not consider this as a major safety concern in this case study.

The software controlling the Therac-25 was written in assembler. The specifications developed here are based mainly on the informal, high-level description given in [7]; they include concurrent and nondeterministic aspects of the machine's behaviour, but they do not include any real-time aspects.

2.2 LOTOS

The reader is referred to the LOTOS standard [6] and [1] for an introduction to LOTOS. LOTOS consists of two parts: so-called *basic* LOTOS and ACT

ONE for abstract data types. Basic LOTOS is very similar to CCS [10], with multi-way synchronisation and some aspects from CSP [5].

An overview of the language, the semantics, some congruences between processes and the notion of testing in LOTOS are given in the following three subsections.

2.2.1 Syntax

Basic LOTOS

LOTOS processes are built up from constant processes, events, and process operators. Events are atomic, indivisible actions. In the following, `P` and `Q` are processes.

operator	**syntax**	**description**
constant	`exit`	successful termination
prefix	`a;P`	prefix `P` by event `a`
choice	`P [] Q`	choice between `P` and `Q`
enable	`P >> Q`	become `Q` after `P` terminates
disable	`P [> Q`	`P` may be interrupted at any time; after interruption, become `Q`
parallel (unconstrained)	`P ||| Q`	`P` in parallel with `Q`
parallel (synchronised)	`P |[l]| Q`	`P` in parallel with `Q`, synchronising on events in list `[l]`
guarded process	`[exp] -> P`	if `exp` holds then become `P`
internal event	`i`	unobservable, internal event
event hiding	`hide E in P`	hide events `E` in `P`

ACT ONE

ACT ONE is the LOTOS sublanguage for specifying abstract data types: values *and* operations. Types are specified using many-sorted equational logic and the semantics are given by *initial* algebras. In this paper we give only 'flat' ACT ONE specifications and do not use any of the specification combining operators of the language.

Full LOTOS

Values and processes are combined in LOTOS in two important ways: values may be associated with events and processes (and process functionality) may be parameterised by values.

We shall use only one form of value association with events: the event offer `a!v` offers value `v` at event `a`. Events with values can only synchronise with *equivalent* values i.e. `a!true` can synchronise with `a!true`, or `a!not(false)`, but not with `a!false`, given the usual theory of Booleans. In the specifications which follow we will use indentation for disambiguation, instead of a proliferation of brackets, when possible.

2.2.2 Semantics

The semantics of a LOTOS specification is given by a structured labelled transition system. Various equivalences and congruences may be defined over such transition systems, and the behaviour expressions which they denote. In this case study, since we are concerned with the observable behaviour of a machine, we will use an observational congruence, *bisumulation* [10], which relates processes that are interchangable in any context. More specifically, we use strong bisimulation because it is used in the LOLA tool (see below); this bisimulation is very similar to the *weak bisimulation* defined in [6], except that the internal event `i` is treated like other events. We do not define this congruence here, but note some of the significant properties of the congruence, as appropriate.

2.2.3 Verification by Testing in LOTOS

Verification in this case study involves proving *temporal properties*. One approach to proving temporal properties of LOTOS specifications is *property* testing [3] using the LITE tool LOLA (*LO*TOS *La*boratory) [11] from the Esprit project LOTOSPHERE.

```
process S[ev1,ev2,ev3,ev4]:exit :=
   (ev1; ev2; ev3; exit) []  (ev1; ev2; ev4; exit)
endproc
```

Figure 1: Specification Process S

```
process T[ev2,ev4,testok]:exit :=
   ev2; ev4; testok; exit
endproc
```

Figure 2: Test Process T

Testing, in LOTOS, is a form of state reachability analysis and it is done in LOLA by expanding a behaviour expression (which may be a parameterised expression), using the appropriate equivalence, to check whether or not a given event occurs on all, none, or some paths, or traces, from the root of the underlying labelled transition system. *Property testing* is done by specifying the given property as a LOTOS test process, concluding with a special test success event(s), and then composing the test process in parallel with the given specification, synchronising on the observable events in the test process (excepting the special test success event(s)). We illustrate this approach with a simple example. Consider the example specification for process `S` given in Figure 1.

Process `S` may perform either `ev1; ev2; ev3; exit` *or* `ev1; ev2; ev4; exit`. Suppose, we wish to ask whether or not the event `ev2` may be followed by the event `ev4` in `S`. We express this *property* by the LOTOS process `T` given in Figure 2. Process `T` has only one possible behaviour: `ev2; ev4; testok; exit`. The last event to occur is the *test event* `testok`.

For property testing, the two processes, that is, the specification process `S` and the test process `T`, are combined together in parallel, synchronising over the events of interest, `ev2` and `ev4`. Then, all possible traces of the combined process are examined to see if the test event occurs; this usually involves algebraic manipulation of the combined process using the laws of the bisimulation. If the test event can be found, then we say that the test has been passed and we can conclude that the behaviour of the test process is a possible behaviour of the specification process.

In this example, the combined process is given in LOTOS by:

```
S |[ev2, ev4]| T.
```

When we expand this process, using the expansion theorem from the bisimulation, one possible trace is `ev1; ev2; ev4; testok; exit` (recall that sychronisation is only required on events `ev2` and `ev4`). Thus the test is passed (for at least one trace), and therefore our specification *does* have the desired property. Obviously, more complex properties are possible; indeed, we can specify any context-free language of traces this way. Moreover, both safety and liveness properties (see Section 3.1) can be tested in this way.

3 Specification I

In this section we proceed to formally specify the high-level behaviour of the Therac-25 in LOTOS. Events are described first, followed by the processes.

There are events for altering the beam intensity and the shield position:

- event `hb` - *high beam*
- event `lb` - *low beam*
- event `hs` - *high shield*
- event `ls` - *low shield*

and there are events for choosing X-ray or electron mode, and for firing the electron beam:

- event `xr` - *X-ray mode*
- event `el` - *electron mode*
- event `fire` - *fire beam*

The 'default' situation is that both the beam and the shield are low, i.e. the machine is ready to irradiate directly with the electron beam. Therefore, in order to operate in X-ray mode, the beam and shield are set to their respective high intensity and position; after firing in X-ray mode, the beam and shield are set back to their respective low intensity and position.

At any point during a radiation treatment, the process may be interrupted and another type of treatment may be chosen or the treatment restarted.

The formal specification is given in Figure 3. The top-level process is `STARTUP` which calls the parameterised process `SETUP` to initialise the machine and set the beam and shield to low, before calling the main process `TREATMENT`. The process `TREATMENT` offers a choice between the X-ray mode, the electron mode, or termination by `exit`. The processes `XRAY` and `ELECTRON` specify the X-ray and electron mode behaviour (respectivel); both processes call the process `TREATMENT` at the end of the 'normal' behaviour, and they may be interrupted, or disabled, by the process `TREATMENT` at any point during the 'normal' behaviour.

In this specification, all events are externally visible (i.e. there are no hidden events). Thus, each process is parameterised by the events occurring within it.

3.1 Verification

Now, we consider the *safety* of the specification `Therac1`. We note that with respect to verification, the word *safety* is overloaded. Here, we specifically mean that life is not endangered. However, we shall only be considering *safety* properties, in the other sense, i.e. properties which state that something bad should *not* happen, as opposed to liveness properties which state that something good *should* happen.

The behaviour which concerns us is the delivery of a radiation overdose and the safety property is given by:

$\boxed{\text{Safety Property I}}$

The machine is unsafe when the electron beam is fired at high intensity and the shield is in the low position.

In order to verify that the specification is safe/unsafe, we must show that Safety Property I does not/does hold. This requires a formalisation of the property with respect to the specified traces, or process prefixes, which would satisfy Safety Property I. We need not consider all traces over all events, but only those traces which are specified behaviours of the machine. Such a trace begins with the initialisation events (i.e. `lb` and `ls` occurring in any order), and an occurrence of `hb` is neither preceded by a `hs` nor succeeded by a `lb` or `hs`. Thus, when the fire event occurs, the beam is high and the shield is low. Formally, these are traces prefixed by traces of the form:

```
specification  Therac1[fire,lb,hb,ls,hs,xr,el] : exit

behaviour
STARTUP[fire,lb,hb,ls,hs,xr,el]
where

process STARTUP[fire,lb,hb,ls,hs,xr,el] :exit :=
   SETUP[lb,ls] >> TREATMENT[fire,lb,hb,ls,hs,xr,el]
endproc

process SETUP[ev1,ev2] :exit :=
   (ev1; exit) ||| (ev2; exit)
endproc

process TREATMENT[fire,lb,hb,ls,hs,xr,el] :exit :=
      (xr; XRAY[fire,lb,hb,ls,hs,xr,el])
   [] (el;ELECTRON[fire,lb,hb,ls,hs,xr,el])
   [] exit
endproc

process ELECTRON[fire,lb,hb,ls,hs,xr,el] :exit :=
   (fire; TREATMENT[fire,lb,hb,ls,hs,xr,el])
      [> TREATMENT[fire,lb,hb,ls,hs,xr,el]
endproc

process XRAY[fire,lb,hb,ls,hs,xr,el] :exit :=
   (SETUP[hb,hs] >> (fire; SETUP[lb,ls])
      >>  TREATMENT[fire,lb,hb,ls,hs,xr,el])
      [> TREATMENT[fire,lb,hb,ls,hs,xr,el]
endproc

endspec
```

Figure 3: Specification I

```
((lb;ls)|(ls;lb)); (not(hb|hs))*; hb; (not(lb|hs|fire))*; fire
```

where we use the notation `*` for zero or more occurrences, `|` for choice, and `not(x|y)` to denote the choice of all events, *excluding* events x and y.

For example, one such trace is: `lb;ls;xr;xr;hb;xr;hb;el;fire`.

Since the traces are described by a regular expression, we can specify them by LOTOS processes and we do so in Figure 4. Process `Nothbhs` corresponds to `(not(hb|hs))*` and process `Notlbhs` corresponds to `(not(lb|hs))*`. The process `TEST` corresponds to `(not(hb|hs))*; hb; (not(lb|hs))*; fire` and `lb;ls;TEST [] ls;lb;TEST` corresponds to the entire trace expression. The event `testok` is the test event.

We used the first of the alternatives as a test process for the specification process `STARTUP`. This process, `UNSAFETEST`, is given as Test Process I in Figure 4. Clearly, if the event `testok` can be reached, then according to Safety Property I, the machine is *unsafe*.

The presentation of the specification in Figure 3 can be deceptive: although it neatly fits on to one page, the use of the disabling operator allows for *many* possible behaviours. The state explosion is quite dramatic and testing by hand is simply not feasible.

LOLA [11] was used to perform the testing, with Test Process I. Since the specification specifies nonterminating processes, an expansion depth is required when performing the tests. A judicious choice of depth proved to be very important: too small and the test was rejected because insufficient process behaviour had been explored; too large and the heap was exhausted. In our case, on a Sun Sparc workstation, the test was rejected when depth<12 and memory was exhausted when depth>13! This experience confirms the experimental and possibly inconclusive nature of testing; *fortunately*, the test passed with depth=12. Thus, we may conclude that the specification `Therac1` is *not* safe.

It is easy to see that one unsafe trace:

```
lb;ls;xr;hb;el;fire
```

is possible by choosing X-ray mode initially, and then interrupting to switch to electron mode before the completion of process `SETUP[hb,hs]`, i.e. after event `hb` but before event `hs`.

This is in fact a behaviour which caused the radiation overdoses in at least two cases, although an element of real-time was involved: the overdose

```
process UNSAFETEST[fire,lb,hb,ls,hs,xr,el,testok] :exit :=
   STARTUP [fire,lb,hb,ls,hs,xr,el]
           |[fire,lb,hb,ls,hs,xr,el]|
   ((lb;ls; TEST[fire,lb,hb,ls,hs,xr,el]) >> testok;exit)
endproc

process TEST[fire,lb,hb,ls,hs,xr,el]:exit :=
   Nothbhs[fire,lb,ls,xr,el]
   >> (hb; Notlbhs[fire,hb,ls,xr,el])
   >> (fire; exit)
endproc

process Nothbhs[fire,lb,ls,xr,el] :exit :=
      fire;Nothbhs[fire,lb,ls,xr,el]
   [] lb;Nothbhs[fire,lb,ls,xr,el]
   [] ls;Nothbhs[fire,lb,ls,xr,el]
   [] xr;Nothbhs[fire,lb,ls,xr,el]
   [] el;Nothbhs[fire,lb,ls,xr,el]
   []exit
endproc

process Notlbhs[fire,hb,ls,xr,el] :exit :=
      ls;Notlbhs[fire,hb,ls,xr,el]
   [] hb;Notlbhs[fire,hb,ls,xr,el]
   [] xr;Notlbhs[fire,hb,ls,xr,el]
   [] el;Notlbhs[fire,hb,ls,xr,el]
   [] exit
endproc
```

Figure 4: Test Process I

occurred only if the operator had switched to electron mode within 8 seconds of starting the X-ray mode. (We note that in [8], the actual sources of error, in these cases, are to be found in the editing routine used by the operator to switch from electron to X-ray mode. See [8] and [12] for descriptions of this fault.)

A simple solution in our specification would be to remove the parallelism in the process `SETUP`, i.e. to replace it by the process:

```
process SETUP[ev1,ev2] :exit :=
   ev1; ev2; exit
endproc
```

When we substituted this process for the original `SETUP` in the specification, and reversed the order of events in the call of `SETUP` from within `XRAY` to `SETUP[hs,hb]`, we found that the test given in Test Process I was rejected (using LOLA) for all expansion depths, until the heap was exhausted. Thus, we have a good idea that this specification is safe. Of course, our tests have not *proven* that is so. We will return to this problem of proving safety in Section 6.

Another solution, the solution proposed by AECL, was to make it 'difficult' for the operator to interrupt the mode from X-ray to electron beam. This was achieved by removing the key cap from the 'up-arrow' key (used to edit the mode data) and covering it with electrical tape!

A further approach to *preventing* the unsafe behaviour in the Therac-25, besides the use of electrical tape, would be to check the status of the beam and shield before firing. LOTOS is ideally suited to modelling this: processes may be parameterised by the status of the beam and the shield. For verification, then, instead of reasoning about the traces leading up to an event, we will 'remember' the current status of the beam and the shield and reason about their state before firing.

We now proceed to parameterise the processes by the status of the beam and shield and to extend the specification to include another aspect of the machine: the reporting of errors. This, in turn, will allow us to uncover a further design feature which can lead to unsafe behaviour.

4 Specification II

Three datatypes, in addition to the usual (library) type boolean, are required in this specification: the type `SHIELD`, the type `BEAM`, and the type `ERROR`.

The specifications of these types are given in Figure 5 and Figure 6.

The type `SHIELD` includes two constants: `up` and `down`, and a test for equality.

The type `BEAM` includes three constants: `high`, `mid`, and `low`, and a test for equality. Three values are included because whilst in operation, the beam intensity was often 'slightly less' than expected, due to the machine being 'out of tune': such a fall in beam intensity could occur up to 40 times a day [7]. A more realistic specification would include many more discrete values for this type (perhaps it should be countably infinite), but for our purposes, one value which is neither `high` nor `low` will suffice.

Finally, we include a type of errors. In the Therac-25, errors are reported by number. We do not know exactly how many different errors were possible, but at least two errors (numbers 53 and 54) are relevant to our specification. Error number 53 denotes that the beam is 'slightly less' than expected, and error number 54 denotes that the beam has high intensity when the shield is in the low position.

The processes are similar to those in the previous specification, with the addition of parameterisation over beam and shield values, and they are given in Figures 7 and 8. But there are a few differences.

The first is that since we are no longer be concerned with traces, but rather with the status of the beam and shield when a `fire` event occurs, the other events (i.e. those which alter the status of the beam and shield, and the choice of xray and electron mode) will be hidden.

The second difference is that for simplicity, we have dispensed with the process `SETUP` and now perform the relevant events sequentially.

The third difference is a rather subtle one. While LOTOS does permit value passing over process enabling (e.g. `P >> accept x:X in Q`), it does not permit value passing over process disabling. This is most unfortunate for us, but perhaps understandable because the semantics of such a construct would be quite complex: the values being passed would have to depend on the point at which the disable occurs. The immediate solution to this problem is to expand processes `ELECTRON` and `XRAY`, using the bisimulation expansion laws. The transformations should remove the disable operators and disabling process `TREATMENT`, and replace them by the appropriate choice of processes which call the disabling process with the appropriate values. But, we cannot get rid of the disable operator altogether, just using the laws of bisimulation. To illustrate the problem, consider the body of the `ELECTRON` process, without any values, and hiding the unobservable events:

```
specification  Therac2[fire,lb,hb,ls,hs,xr,el] :exit(beam,shield)

library boolean endlib

type SHIELD is boolean
sorts shield
opns  up, down  : -> shield
_eq_ : shield, shield -> bool
eqns
ofsort bool
   up eq down   = false;
   up eq up     = true;
   down eq down = true;
   down eq up   = false;
endtype

type BEAM is boolean
sorts beam
opns high, mid, low : -> beam
   _eq_ : beam, beam -> bool
eqns
ofsort bool
   high eq low  = false;
   high eq high = true;
   low eq low   = true;
   low eq high  = false;
   high eq mid  = false;
   low eq mid   = false;
   mid eq low   = false;
   mid eq mid   = true;
   mid eq high  = false;
endtype
```

Figure 5: Beam and Shield Datatypes

```
type ERROR is boolean
sorts errnum
opns  err53, err54  : -> errnum
endtype
```

Figure 6: Error Datatype

```
(fire; TREATMENT[fire]) [> TREATMENT[fire]
```

This process is expanded to:

```
   TREATMENT[fire]
[] (fire; (TREATMENT[fire])[> TREATMENT[fire])
```

Because `TREATMENT` is recursive and disabled by itself, each expansion of `TREATMENT[fire]` introduces a further `[> TREATMENT[fire]`. We do not have a law `P [> P = P` because, in general, `P [> P` is *not* bisimular to `P`. Certainly it does not hold for finite processes; consider, for example, the process `process P := a;b;exit`. However, in our specification, because `TREATMENT` is recursive and it is disabled by itself in the appropriate way, we do have an equivalence between `TREATMENT[fire] [> TREATMENT[fire]` and `TREATMENT[fire]`. Thus we are able to transform Specification I into Specification II which includes parameterised processes.

The final difference concerns the event `fire`. In processes `ELECTRON` and `XRAY`, the event `fire` is replaced by a process `FIRE`. This process is also parameterised by a beam and shield value and 'traps' the unsafe situations by calling the process `ERROR`, with an error number; this process terminates after reporting the error. The `ERROR` process may also be called when the machine is 'out of tune'. Note that there is an element of nondeterminism in the reporting of this error.

4.1 Verification

Again, the behaviour which concerns us is the delivery of a radiation overdose. This is formalised very concisely by the event `fire!high!down`. If there are traces which include this event, then according to Safety Property I, the machine is unsafe.

```
behaviour
STARTUP[fire]
where

process STARTUP[fire] :exit(beam,shield) :=
   hide lb,ls in
   lb; ls; TREATMENT[fire](low,down)
   endproc

process TREATMENT[fire](b:beam,s:shield):exit(beam,shield):=
hide xr,el in
      (xr; XRAY[fire](b,s))
   [] (el;ELECTRON[fire](b,s))
   [] exit(b,s)
endproc

process ELECTRON[fire](b:beam,s:shield) :exit(beam,shield) :=
     (Fire[fire](b,s) >> TREATMENT[fire](b,s))
   [] TREATMENT[fire](b,s)
endproc

process XRAY[fire](b:beam,s:shield) :exit(beam,shield) :=
hide lb,hb,ls,hs,xr,el in
     TREATMENT[fire](b,s)
  [] hb; (TREATMENT[fire](high,s)
        [] hs;(TREATMENT[fire](high,up)
           [] (Fire[fire](high,up) >>
                        (TREATMENT[fire](high,up)
                       [] lb; (TREATMENT[fire](low,up)
                            [] ls; TREATMENT[fire](low,down)
                              )
                        )
                 )
                 )
          )
endproc
```

Figure 7: Specification II

```
process FIRE[fire](b:beam,s:shield) :exit :=
   hide err in
      [(b eq high) and (s eq down)] -> ERROR[err](err54)
   [] [not(b eq high)] ->  ZAP[fire](b,s)
   [] [not(b eq high) and not(b eq low)] ->  ERROR[err](err53)
endproc

process ZAP[fire](b:beam,s:shield) :exit:=
   fire!b!s; exit
endproc

process ERROR[err](e:errnum) :exit:=
   err!e; exit
endproc

endspec
```

Figure 8: Specification II

```
process UNSAFETEST[fire,testok](b:beam,s:shield):exit(beam,shield):=
   TREATMENT[fire](low,down)
      |[fire]|
   UNSAFETEST[fire,testok](low,down)
endproc

process UNSAFETEST[fire,testok](b:beam,s:shield):exit(beam,shield) :=
   fire!high!down; testok; exit(any beam, any shield)
endproc
```

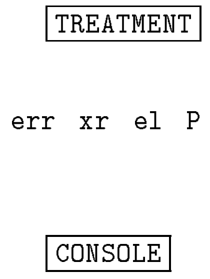
Figure 9: Test Process II

The LOTOS test process which tests for this event, Test Process II, is given in Figure 9. When we used LOLA with Test Process II, all (finite) tests were rejected. Thus, we have *some* confidence that this is, at least, a *safer* specification.

However, the story does not end here. So far, we have neglected to specify the machine operator - a key player in the story. In the next section we add the specification of the operator.

5 Specification III

In this specification, the operator is specified by the process `CONSOLE`. The overall behaviour of the system is given by machine initialisation followed by the process `CONSOLE` combined in parallel with the process `TREATMENT`, synchronising over events `err`, `xr`, `el` and `P`. These are the only observable events which the operator may engage in, i.e. the operator selects the X-ray or electron mode, or takes action after acknowledging an error message on the screen.



We do not attempt to specify the complete behaviour of the operator, but only that part which is relevant here. As mentioned above, the operator has the choice of choosing the X-ray or electron modes, or of finishing the treatment (i.e. `exit`). However, in addition, the operator can press the 'P' (for 'pause') key, after an error occurs, to resume the treatment. This is modelled in the LOTOS process by the event `P`.

In the new specification of the process `ERROR`, treatment, i.e. the beam is fired, is continued *only* after the `P` event. But since the combined overall process must synchronise, or agree, on both this event and the error event, if the operator does not also offer the same error event and value (i.e. acknowledgement of the error - this is equivalent, here, to displaying the error on the screen) and the event `P`, then the process will be stopped, or deadlocked.

Errors in the Therac-25 are reported by number, eg. as 'Malfunction 54', or 'Malfunction 53'. Thus, the error message does not reflect the nature, nor the relative importance or priority of the message; the nondeterministic choice between error offers in process `CONSOLE` relects this aspect.

The processes `TREATMENT`, `ELECTRON`, `XRAY`, `FIRE`, and `ZAP` remain unchanged. Specifications for the processes `STARTUP`, `ERROR`, and `CONSOLE` are given in Figure 10.

5.1 Verification

Again, the behaviour which concerns us is the delivery of a radiation overdose, i.e the event `fire!high!down`. If there are traces which include this event, then according to Safety Property I, the system is unsafe.

We used the same LOTOS test process again, i.e. Test Process II given in Figure 9, but with the new specifications of `STARTUP`, `ERROR`, and `CONSOLE` given in Figure 10. When we used LOLA to check whether or not event `testok` could be reached, tests with expansion depths up to 8 were rejected,

```
process STARTUP[err,fire,xr,el,P] :exit(beam,shield) :=
   hide lb,ls in
   lb; ls;
      (TREATMENT[err,fire,xr,el,P](low,down)
           |[xr,el,err,P]|
       CONSOLE[err,fire,xr,el,P])
endproc

(*firing proceeds after agreement on the error followed*)
(* by agreement on P key *)
process ERROR[err,fire,P](e:errnum,b:beam,s:shield) :exit :=
      err!e; P; ZAP[fire](b,s)
endproc

process CONSOLE[err,fire,xr,el,P] :exit(beam,shield) :=
      xr; CONSOLE[err,fire,xr,el]
   [] el; CONSOLE[err,fire,xr,el]
   [] err!err53; P; CONSOLE[err,fire,xr,el]
   [] err!err54; P; CONSOLE[err,fire,xr,el]
   [] exit(any beam, any shield)
endproc
```

Figure 10: Specification III

but the test *did* pass with depth=9. Thus the specification of the machine, in parallel with the operator, is *unsafe*.

The source of the problem of course lies with the operator, and more specifically, with the user interface. Since some 'trivial' errors could occur up to 40 times a day, operators quickly became used to overriding all errors with the 'P' key, regardless of the error number. Indeed, in [7], it is reported that one patient was fatally overdosed *three* times in this way.

A final twist to this tragic story concerns yet another equipment failure. The clinic (where the first fatality occurred) kept the operator and patient in separate, heavily shielded rooms, in order to protect the operator from the radiation. Communication between the two rooms was possible via an intercom and closed-circuit television. However, on the day of the overdose(s), the intercom and television were not working.

6 On Proving Safety

Testing proved to be an adequate verification technique for Specifications I and III because the specifications were unsafe, i.e. they *passed* the unsafe test. However, testing only provides a semi-decision procedure when the processes under investigation are infinite: in these cases, test rejection proves nothing conclusive as we can only try a finite number of test depths. Test rejection *suggests* that the specifications are safe, but how could we prove this to be so?

One way is to prove that the test is never passed. This may be done by transforming the combined specification and test process into a set of recursive equations and then examining the non-recursive parts to ensure that the test event does not occur.

6.1 An Example: Proving the test is never passed

As an example of a specification we believe to be safe, consider the specification discussed in Section 3.1; namely, a transformed version of Specification I, modified to exclude the parallelism in `SETUP`. This specification is given as Specification IV in figure 11.

Now consider testing Specification IV with Test Process I, namely `UNSAFETEST[lb,ls,hb,hs,xr,el,fire]`.

We expand this process using the bisimulation laws which distribute parallelism through choice, and represent deadlock by the internal event `i`

```
behaviour
STARTUP[lb,ls,hb,hs,xr,el,fire]
where

process STARTUP[lb,ls,hb,hs,xr,el,fire] :exit :=
   lb; ls; TREATMENT[lb,ls,hb,hs,xr,el,fire]
   endproc

process TREATMENT[lb,ls,hb,hs,xr,el,fire]:exit:=
      (xr; XRAY[lb,ls,hb,hs,xr,el,fire])
   [] (el;ELECTRON[lb,ls,hb,hs,xr,el,fire])
   [] exit
endproc

process ELECTRON[lb,ls,hb,hs,xr,el,fire]fire]:exit :=
     (fire; TREATMENT[lb,ls,hb,hs,xr,el,fire])
   [] TREATMENT[lb,ls,hb,hs,xr,el,fire]
endproc

process XRAY[lb,ls,hb,hs,xr,el,fire]:exit :=
     TREATMENT[lb,ls,hb,hs,xr,el,fire]
  [] hs; (TREATMENT[lb,ls,hb,hs,xr,el,fire]
        [] hb;(TREATMENT[lb,ls,hb,hs,xr,el,fire]
           [] (fire; TREATMENT[lb,ls,hb,hs,xr,el,fire]
                       [] lb; (TREATMENT[lb,ls,hb,hs,xr,el,fire]
                            [] ls; TREATMENT[lb,ls,hb,hs,xr,el,fire]
                               )
                 )
                 )
          )
endproc
```

Figure 11: Specification IV

followed by `stop` (eg. `a:P |[a,b]| b:Q = i;stop`). We use ellipsis notation to abbreviate a process whose form is not relevant to the rest of the transformation, and the process `TESTPROC` to abbreviate part of the test process, namely `TEST[lb,ls,hb,hs,xr,el,fire] >> testok; exit`.

```
UNSAFETEST[lb,ls,hb,hs,xr,el,fire]
=lb; ls;
     TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
     TESTPROC
```

where

```
TREATMENT[lb,ls,hb,hs,xr,el,fire]
   |[lb,ls,hb,hs,xr,el,fire]|
 TESTPROC
=  (i; stop)
[] (xr;
    XRAY[lb,ls,hb,hs,xr,el,fire]
       |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC)
[] (el;
    ELECTRON[lb,ls,hb,hs,xr,el,fire]
       |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC).
```

Expanding the second choice gives:

```
XRAY[lb,ls,hb,hs,xr,el,fire]
       |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC
= TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC
[] hs; (...)
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC
= TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC
[] i; stop
```

and expanding the third choice gives:

```
ELECTRON[lb,ls,hb,hs,xr,el,fire]
       |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC)
= TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
  TESTPROC
[] fire;  TREATMENT[lb,ls,hb,hs,xr,el,fire]
              |[lb,ls,hb,hs,xr,el,fire]|
          TESTPROC
= TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC
[] TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC
= TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC.
```

And so we are left with:

```
UNSAFETEST[lb,ls,hb,hs,xr,el,fire,testok]
=lb; ls;
     TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
     TESTPROC
```

and

```
TREATMENT[lb,ls,hb,hs,xr,el,fire]
   |[lb,ls,hb,hs,xr,el,fire]|
 TESTPROC
=  (i; stop)
 [] TREATMENT[lb,ls,hb,hs,xr,el,fire]
      |[lb,ls,hb,hs,xr,el,fire]|
   TESTPROC.
```

Now we have a set of recursive equations in which the test event `testok` cannot not occur and so we conclude that the test cannot be passed.

This approach is dependent upon finding the right set of recursive equations (in this case it was necessary to eliminate the `[>` operators). An alternative approach to proving safety involves the use of temporal logic.

6.2 Temporal Logic

Since the properties which interest us are essentially temporal properties, another verification technique involves expressing the properties in temporal logic [9]. When the property is specified in temporal logic, then the verification problem becomes that of showing that the temporal formula for the property is satisfied by the LOTOS specification. Temporal formulae include the usual first order connectives, as well as temporal operators. The temporal operators which we use are:

- $\circ$ - *next*
- $\Diamond$ - *sometime*
- $\Box$ - *always*
- $\mathcal{W}$ - *until*

Informally, $\circ P$ means that the formula P is true in the next state; $\Diamond P$ means that the formula P is true in one of the subsequent states; $\Box P$ means that the formula P is true in all of the subsequent states; and $P \mathcal{W} Q$ means that the formula P is true in all subsequent states, or it is true in all subsquent a states until a state is reached in which Q is true.

For example, the verification property for Specification I given in Section 3.1 is specified by: $\neg\Diamond(P1 \wedge \circ\circ P2)$, or equivalently, by $\Box\neg(P1 \wedge \circ\circ P2)$, where

$$\begin{aligned} P1 &= (lb \wedge \circ\ ls) \vee (ls \wedge \circ lb) \\ P2 &= (\neg(hb \vee hs))\ \mathcal{W}\ P3 \\ P3 &= hb \wedge \circ P4 \\ P4 &= (\neg(lb \vee hs))\ \mathcal{W}\ fire \end{aligned}$$

The verification property for Specifcation III is more concisely given by: $\neg \diamond fire!high!down$.

One way to prove satisfaction is to give a temporal logic semantics to LOTOS: then the problem is reduced to showing that the temporal formula for the LOTOS specification implies the temporal formula for the given property.

In [4], a temporal logic semantics for *basic* LOTOS is given. The semantics is denotational in style and respects trace equivalence; this is adequate for proving our safety properties, athough it is too weak for proving liveness properties. Given this semantic function, $\mathcal{L}$, in order to show that a safety property P, expressed as a temporal formula, holds for a LOTOS specification S, we need to show $\mathcal{L}(\mathrm{S}) \Rightarrow \mathrm{P}$.

The semantic function $\mathcal{L}$ is relatively straightforward for event prefixing and choice, but becomes quite complicated for the other operators. Because of this complexity, we do not give the semantics of our specifications here, but we give just a flavour of the semantics by giving an expression for the semantics of the body of the `TREATMENT` process. Processes are not assumed to be *closed* and so the event e is used to denote interaction with an environment. The semantics is given by the disjunction:

$$e\mathcal{W}(xr \wedge \circ\mathcal{L}(\texttt{XRAY})) \vee e\mathcal{W}(el \wedge \circ\mathcal{L}(\texttt{ELECTRON})) \vee e\mathcal{W}(d \wedge \circ(e\mathcal{W}false))$$

The full semantics for the `TREATMENT` process is given by a maximal fixed point of a recursive equation, i.e. the equation for $\mathcal{L}(TREATMENT)$ has the form

$$\nu x.e\mathcal{W}(xr \wedge \ldots \circ x) \vee e\mathcal{W}(el \wedge \ldots \circ x) \vee e\mathcal{W}(d \wedge \circ(e\mathcal{W}false))$$

where ν is the maximal fixpoint operator.

There are three difficulties with this approach. The first is that the formulae are very large and unwieldy; rigorous proofs by hand are unmanageable. There is hope though, as the authors of [4] do state that an automated tool for temporal logic proof is under construction. The second difficulty concerns the fact that validity in the logic is not decidable, and so a decision procedure for the entire logic will never be possible. Finally, the third difficulty is that the current semantics does not cover full LOTOS and so specifications such as Specification III are excluded from consideration, until such time as the semantics is extended to include data types. Another possibility, yet to be explored, is to carry out model-checking in a branching-time temporal logic.

7 Discussion

From an informal description of the Therac-25 radiation machine, a formal specification in LOTOS was defined (Specification I). This specification was concise enough to fit onto one page, yet still captured the essential behaviour of the machine, including a fatal design flaw.

A safety property was formulated (Safety Property I) and the verification of the property explored by considering the traces which could lead to the delivery of a radiation overdose. These traces, or behaviours, were specified as LOTOS processes and using the LOLA tool, we showed that the specified machine *could* deliver a radiation overdose. We did not prove that this specification was a most general set of unsafe traces because we did show that the machine was unsafe. When the converse is the case, then we must be careful to show that we are considering the most general specification of unsafe processes. Furthermore, we considered only traces which were specified as valid machine behaviours. A more comprehensive approach to safety would be to consider *all* traces leading to unsafe behaviour and then relate them to the specified behaviour in order to construct a fault-tolerant specification.

Further specifications (Specifications II and III) allowed a much simpler formalisation of Safety Property I because additional, redundant information was added to the specification in the form of values representing the state of the beam and the shield. The behaviour of Specification II changed dramatically when it was combined with the `CONSOLE` process in Specification II, demonstrating the importance of considering the behaviour of the environment of the machine (or, indeed, any process) when proving safety properties.

Several important aspects of the disable operator `[>` were uncovered in this case-study. This very powerful operator prevented a seemingly simple transformation from a specification without values to one with values. This does suggest that it should be introduced into a specification with great care, and that further properties of `[>`, w.r.t. the various congruences and equivalences, should be studied (eg. under what conditions does `P [> P = P` hold?).

The testing tool LOLA was used extensively, and successfully, but the need for a full temporal logic semantics and an automated theorem prover or model checker was identified.

7.1 Conclusions

We have achieved our goals in this small case study: to consider the formal specification and verification of a safety-critical medical application. LOTOS has proved to be an appropriate specification language and property testing, using LOLA, was a valuable verification tool. Since our specifications are of potentially infinite processes, testing was only conclusive when the test is passed, i.e. the specification is unsafe, as was the case for the main specifications (Specifications I and III). As an experiment, we presented these specifications to several Computing Scientists for informal verification; in nearly all cases, the unsafe behaviour was not detected. Thus, we are convinced that (some) formal verification is necessary when considering applications and problems of this nature.

The problem of proving safety is more difficult, but we also showed how testing and temporal logic may be used in a proof of safety.

We are aware that this case study has not delved deeply into safety issues of a realistic application and presents a rather simplistic view. A more realistic view would at least involve safety requirements at different levels, for different components of the system. For example, there might be some modes of behaviour for which a state with a high intensity beam and low shield is required.

Finally, the interested reader is encourage to read the detailed report by Leveson and Turner [8] (which appeared after this specification experiment had been carried out).

7.2 Future Work

The fundamental question is could formal methods have helped to prevent the Therac-25 tragedy, and how can they be used in future the development of reliable safety-critical software? Formal methods are *not* a panacea, but they must have an important role to play. Future work is planned and will concentrate on the development of further tools and techniques as discussed above, and also on a coherent approach to the application of formal methods within the established framework for building safety-critical systems in engineering and science.

Acknowledgements

Carron Kirkwood repeated some of the verification tasks using the PAM (Process Algebra Manipulator) tool and in the course of her work she uncovered some of the problems with the disable operator as well as several typos in an earlier draft. An anonymous reviewer also made several useful suggestions for improvement.

References

[1] T. Bolognesi, E. Brinksma, Introduction to the ISO Specification Language LOTOS, Computer Networks ISDN Systems **14**, pp. 25-59,1987.

[2] Radiation Accident at Hammersmith, British Medical Journal, number 5507, 23 July, pg. 233, 1966.

[3] F.J. Carrasco, J.J. Gil, A Method for Specifying and Validating Communication Protocols in LOTOS, in Formal Description Techniques V, M. Diaz, R. Groz (eds.), North-Holland, 1993.

[4] A. Fantechi, S. Gnesi, C. Laneve, An Expressive Temporal Logic for Basic LOTOS, in Formal Description Techniques II, S. Vuong (ed.), North-Holland, 1990, pp. 261-276.

[5] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall International, 1985.

[6] ISO [ISO:8807] *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

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

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

[9] Z. Manna, A. Pnueli, Verification of Concurrent Programs: The Temporal Framework, in R.S. Boyer, J.S. Moore (eds.) Correctness Problem in Computer Science, Academic Press, 1981, pp. 215-273.

[10] R. Milner, Communication and Concurrency, Prentice-Hall International, 1989.

[11] J. Quemada, S. Pavón, A. Fernandez, Transforming LOTOS specifications with LOLA - The Parameterised Expansion, in Formal Description Techniques I, K. Turner (ed.), North-Holland, 1988.

[12] M. Thomas, A Proof of *Incorrectness* using LP: The Editing Problem in the Therac-25, High Integrity Systems Journal, Volume 1, Issue 1, OUP, 1993.