Kenneth J. Turner and Qian Bing. Protocol Techniques for Testing Radiotherapy Accelerators. In Moshe Vardi and Doron Peled, editors, Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV), Copyright Springer-Verlag, Berlin, November 2002.

Protocol Techniques for Testing Radiotherapy Accelerators

Kenneth J. Turner and Qian Bing

Computing Science and Mathematics, University of Stirling, Scotland FK9 4LA
`kjt@cs.stir.ac.uk` and `qb@cs.stir.ac.uk`

Abstract. The nature of radiotherapy accelerators is briefly explained. It is argued that these complex safety-critical systems need a systematic basis for testing their software. The paper describes a novel application of protocol specification and testing methods to radiotherapy accelerators. An outline specification is given in LOTOS (Language Of Temporal Ordering Specification) of the accelerator control system. It is completely infeasible to use this directly for test generation. Instead, specification inputs are restricted using annotations in a Parameter Constraint Language. This is automatically translated into LOTOS and combined with the accelerator specification. It then becomes manageable to generate tests automatically of the actual accelerator to check that it agrees with its specification according to the relation *ioconf* (input-output conformance). Sample input annotations, their translation to LOTOS, and the resulting test cases are described.

1 Introduction

This paper presents a novel application of existing protocol specification and testing techniques to a new domain. Radiotherapy accelerators are complex, software-controlled, safety-critical systems. It is highly desirable to test their control systems systematically using test suites generated automatically from formal specifications. The work demonstrates that LOTOS can be successfully used to specify accelerators. The paper focuses on making automated test generation practicable for such specifications.

1.1 Radiotherapy Accelerators

Radiotherapy equipment is used medically to deliver controlled doses of radiation to a patient, usually to destroy cancerous tissue. Among the several kinds of radiotherapy equipment, the most important is the linear accelerator ('accelerator' or 'linac'). This is so-named because it accelerates a beam of electrons to high energy that can be used directly or to generate x-rays. Accelerators are highly specialised pieces of equipment that require special housing and trained operators. For this reason, they are generally found in oncology (cancer) clinics. As an indication, about 300 accelerators are currently used in the UK. Some countries such as the US make more extensive use of accelerators.

Radiotherapy is a safety-critical procedure that demands accurate delivery of radiation. A number of radiation accidents have been well documented (e.g. [13, 14]). The Therac-25 accelerator is infamous as having caused accidental injuries, in some cases leading to death [15]. In fact, a radiation underdose is as undesirable as an overdose since it may fail to kill a tumour. Not delivering radiation to the exact area is also serious as it damages surrounding healthy tissue instead of destroying cancerous growth.

Radiotherapy equipment is regulated, designed and tested to very high standards. A review of standards for software-controlled medical devices is given in [11]. The main international standards of relevance to this paper are those in the IEC 601 series. This is a very large collection of standards, specifically including programmable electrical medical systems. A number of subsidiary standards concern accelerators [7]. The US Food and Drug Administration has published guidelines on Good Manufacturing Practice [3] that are relevant to software-controlled medical devices. Radiotherapy machines are typically certified in the US before they are sold anywhere in the world. The American Association of Physicists in Medicine has laid down a code of practice specifically for radiotherapy accelerators [16]. The Canadian Atomic Energy Authority also plays an active role in regulating radiotherapy devices. The European Commission is defining standards for safety of medical equipment (e.g. the Medical Devices Directive [2]). More general software development standards are also relevant, such as the ISO/IEC 9000 series on quality assurance and its European EN equivalents.

Early radiotherapy equipment was essentially hardware. Hardware aspects of accelerators are regularly and thoroughly checked. For example, dosimeters (dosage meters) are periodically calibrated against national standards. The accuracy of radiation delivery is also regularly checked in simulated treatments. The hardware is extensively protected by interlocks that address situations like power supply failure, dosimeter failure, or entry to the treatment room during radiation delivery.

Accelerators are, however, complex software-controlled systems. (The Therac-25 accidents stemmed in part from a reduction in the number of hardware interlocks.) Accelerator software resembles standard application software. It requires a graphical user interface, peripheral input-output, file system operations, and data communications. The accelerator software depends on a conventional style of operating system. The software must respect strict demands for dependable, real-time operation. Software, unlike hardware, does not deteriorate over time so that different reliability concerns apply. Like any application, the accelerator control software is upgraded from time to time by the manufacturers. Of course, the software is developed much more carefully than conventional application software. However with new accelerator software, it is desirable to check that the new version has not introduced any flaws. Surprisingly, there seems to be little automation to help clinics to do this.

1.2 Applicability of Formal Methods

Formal methods are an obvious choice to support the development and testing of radiotherapy equipment. Somewhat unexpectedly, radiotherapy equipment has attracted little attention from the formal methods community. [18] is one of few contributions, having made use of LOTOS (Language Of Temporal Ordering Specification [8]) to show how the Therac-25 flaws could have been identified. The only other work known to the authors uses Z to specify the design of software for a radiation therapy machine [10].

Conformance testing uses experimentation to check an implementation against its formal specification. Tests are derived from the specification, then applied to the Implementation Under Test. Based on observations made during test execution, a verdict is given about the correct functioning of the implementation. The unique contribution of this paper is the application to radiotherapy accelerators of techniques normally used only with protocols: LOTOS, and methods for generating conformance tests.

An LTS (Labelled Transition System) gives the semantics of a LOTOS specification. The accelerator implementation is presumed to be modelled by an IOLTS (Input-Output Labelled Transition System). Conformance of an implementation can then be expressed with respect to its specification using a formal relation. One such relation, *ioconf* (input-output conformance [19]), is used as the criterion for correct accelerator design. The test suite for an accelerator is generated from a LOTOS specification following an algorithm based on that given in [19]. The authors have programmed CADP (Cæsar Aldébaran Development Package [4]) to generate accelerator test suites automatically. Each test case in the generated test suite defines possible inputs and expected outputs.

The work reported in this paper concentrates on automated test generation. This is of greatest practical value to oncology centres. The approach adopted currently focuses on general behavioural correctness of the accelerator control system. The generated test suites can detect flaws such as failing to reject invalid input values, permitting incorrect input sequences, or sending wrong commands to the accelerator hardware. Although accelerators have real-time behaviour, the current model uses only an abstract notion of time. The test suites cannot therefore detect timing problems such as delivering radiation at the wrong absolute rate, or for too long in absolute terms.

2 Radiotherapy Accelerators

2.1 Accelerator Hardware

The entire accelerator is located in a treatment room. This is heavily screened to prevent radiation leakage to the outside. Access is via an interlocked door (or gate) from the control room. The control room houses the operator console and the supporting computer systems. For security, the operator must insert a key into the console before it will work. [5] is a comprehensive introduction to the theory and practice of accelerators.

A typical accelerator is shown schematically in figure 1. The accelerator proper is mounted on a gantry that rotates about the horizontal axis. The accelerator uses a travelling waveguide to accelerate electrons from an electron gun. The beam is controlled so as to yield electrons with energies typically in the range 6 to 20 MeV (million electron-volts). Radiation dosages are measured in MUs (monitor units). MUs reflect the calibration of dosimeters rather than any absolute unit, but 1 MU approximates to 1 cGy (centigray, a standard unit of radiation dosage).

The horizontal electron beam is bent by magnets through 90°(or 270°) so that it points downwards. In electron mode the electrons emerge through a radio-transparent plate to reach the patient. In x-ray mode the electrons strike a target, causing a shower of x-rays towards the patient.

The treatment head contains a collimator. This consists of four movable plates, two that move in the X direction and two that move in the Y direction. They define a rectangle that restricts the beam to a defined aperture. A sophisticated accelerator will have an MLC (multi-leaf collimator). This has many (one or two hundred) individually movable leaves that may be used to set an arbitrary shape for the beam aperture. An ‘accessory’

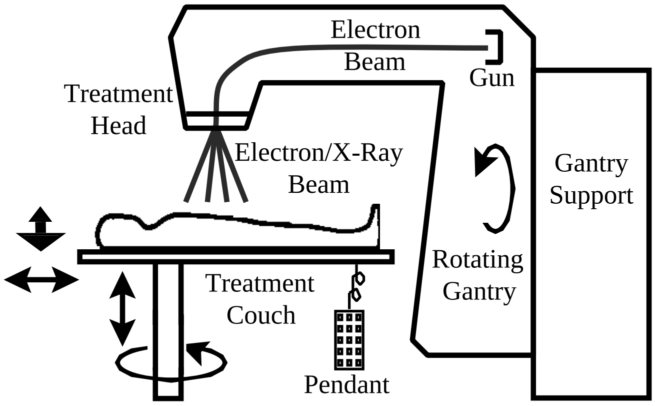

Fig. 1. Accelerator Outline

may be fitted to the treatment head to control the beam distribution. The treatment head also houses an optical system that allows the shape and position of the beam to be seen on the patient's skin prior to treatment.

The patient lies on a treatment couch that may be adjusted for height, in-out position (longitude), side-to-side position (latitude), and rotation. A pendant (remote control device) is attached to the couch for setting the couch position and also for rotating the gantry. The operator sets up the patient and the accelerator so that the correct part of the body will be irradiated.

2.2 Accelerator Control System

During treatment, the delivered radiation dose is read periodically from the accelerator. For safety, this is measured by two independent dosimeters whose readings are accumulated. The first dosimeter reading usually decides when treatment is complete. The accumulated dose should rise to the planned dose, but some tolerance is allowed. In case the first dosimeter does not work properly, readings from the second one are used as a backstop. Treatment is aborted if the second dose measurement exceeds the planned dose by 20 MUs or more. The dose rate is also checked at every measurement. It may not deviate from the planned rate by more than an amount that depends on the particular treatment. Finally, the treatment time is calculated from the dose and dose rate. A clock is read to ensure that treatment does not exceed the planned time by more than 10%.

Figure 2 shows the main elements of a typical control system. The arrows shows the direction but not details of information flow. The operator usually starts by arranging the patient and the accelerator geometry in the treatment room. The in-room computer displays setup information in the treatment room. The operator then retires to the control room, where treatment details are set up on the console computer. The console display shows the current accelerator setup and status. Treatments are usually planned separately and stored on a file server. The treatment is downloaded to the treatment computer and thence to the console computer.

Peripheral input-output is handled by a separate communications computer. During actual treatment (delivery of radiation), the accelerator is under the command of a sep-

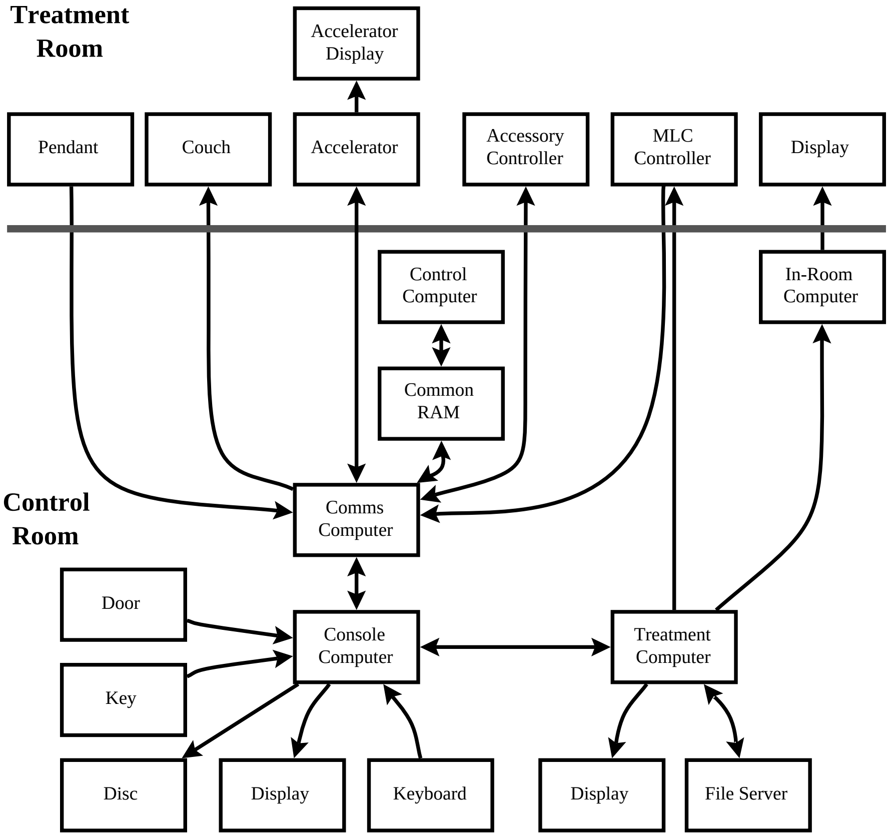

Fig. 2. Accelerator Control System

arate control computer that issues commands and monitors status via common RAM. The entire system uses about half a dozen computers or microprocessors so it is not surprising that the software is complex.

For the work reported in this paper, the control system has been simplified as shown in figure 3. The detailed information flows are shown against the arrows. This effectively groups all of the control functions in a single black box, with the main inputs and outputs as shown. Although the real system involves considerable communication among subsystems, figure 3 is a legitimate abstraction since it shows only the externally observable interfaces. The clock abstracts the real-time aspects of processing.

3 Accelerator Specification

The simplified accelerator control system shown in figure 3 has been specified in LOTOS. Although the specification reflects a particular type of accelerator, the description is typical of a range of accelerators. The specification is straightforward: 730 non-comment lines, about half of these devoted to data types. The outline specification struc-

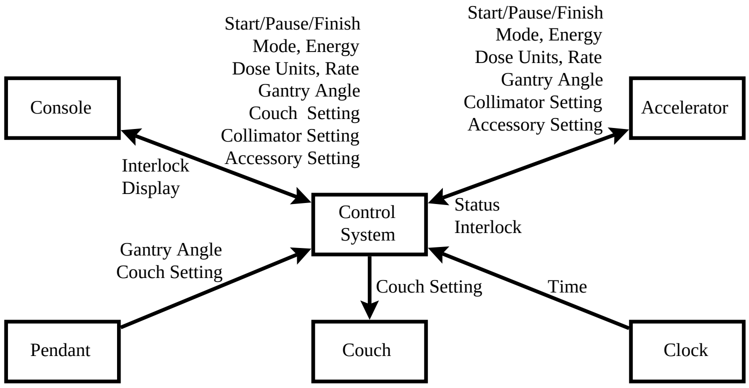

Fig. 3. Simplified Accelerator Control System

ture appears below. The gates correspond to the control system interfaces in figure 3. The specification was checked by simulation, and by evaluation of typical scenarios drawn from the project's medical partner. The complete specification is given in [17].

```
specification System [console, pendant, accelerator, couch] : noexit
  library ...                                                (* library types *)
  type Time is ...                                            (* time values *)
  type Constants is ...                                  (* system constants *)
  type Signal is ...                                    (* subsystem signals *)
  type Interlock is ...                               (* hardware interlocks *)
  type Angle is ...                                   (* gantry/couch angle *)
  type Mode is ...                                       (* treatment mode *)
  type Energy is ...                                        (* beam energy *)
  type Dose is ...                                  (* radiation dose units *)
  type Rate is ...                                   (* radiation dose rate *)
  type Position is ...                                   (* couch position *)
  type NaturalOperations is ...                    (* operations on naturals *)
  type Collimator is ...                              (* collimator setting *)
  type Accessory is ...                                (* accessory setting *)
  type Setting is ...                                (* accelerator setting *)
  type Status is ...                                  (* accelerator status *)
  type Couch is ...                                        (* couch setting *)
  type Display is ...                                    (* console display *)
behaviour                                              (* overall behaviour *)
  hide time in                                             (* internal gate *)
    Control [console, pendant, accelerator, couch, time]  (* control system *)
      (DefaultSetting, DefaultCouch)
  |[time]|                                       (* synchronised on times *)
    Clock [time] (0)                                        (* system clock *)
where
  process Clock [time] (Time) ...                           (* system clock *)
  process Mode [console] (Setting) ...                      (* mode setting *)
  process Energy [console] (Setting) ...                  (* energy setting *)
  process Dose [console] (Setting) ...                      (* dose setting *)
  process Rate [console] (Setting) ...                                   (* rate setting *)
  process Gantry [accelerator, pendant] (Setting) ...                    (* gantry setting *)
  process Collimator [console] (Setting) ...                             (* collimator setting *)
  process Accessory [console] (Setting) ...                              (* accessory setting *)
  process Couch [pendant] (Couch) ...                                    (* couch setting *)
  process Control [console, pendant, accelerator, couch, time]           (* control system *)
  (Setting, Couch) ...
  process Setting [console, pendant, accelerator, couch]                 (* accelerator setting *)
  (Setting, Couch) ...
  process Monitoring [console, accelerator, time]                        (* treatment monitoring *)
  (Setting, Couch, Status, Time) ...
```

Many of the data types simply rename the natural numbers (e.g. dose units, angles, positions). Although in practice these parameters are floating point numbers with various scales and ranges, this simplified approach is acceptable. It just means that the offset and units for these parameters are calculated differently from normal.

Although the specification contains a *Clock* process, this merely increments a time count. It would be necessary to use E-LOTOS (Enhanced LOTOS [9]) if a more precise notion of time were required. However, E-LOTOS tool support is still rather incomplete. The current approach therefore deals with only an abstract notion of time.

The main process is *Control*. Initially this allows accelerator setup by the *Setting* process. Setting the gantry or the couch position causes movement commands to be issued, but other inputs are merely stored prior to treatment. The console display is updated after every input to reflect the current accelerator status. The operator may initiate treatment once a valid set of parameters has been entered.

The accelerator setting is then sent to the accelerator and radiation begins. The *Monitoring* process periodically reads the accelerator status, i.e. the two dosimeter readings. Normally, treatment continues until the prescribed dose has been delivered. Section 2.2 describes how the accumulated doses and the clock are used to terminate treatment. The operator is permitted to pause and resume treatment, perhaps because the patient is restless. Any abnormal condition such as an interlock stops the treatment immediately. On completion of treatment, radiation is stopped and a *Finished* signal is sent to console.

4 Test Constraints

4.1 Test Strategy

The aim is to produce useful system tests from an accelerator specification. There does not seem to be a systematic procedure for clinics to check accelerator software. Automated testing can therefore supplement normal clinical practice, particularly following a software upgrade. The test suite automatically generated from an accelerator specification can be converted into the same format as is used for patient treatments. It can then be loaded into the treatment computer and executed automatically.

There is a choice of when to generate tests. Ideally, a symbolic transition system would first be created from the specification. (This would have transitions labelled like *gate ?value:Sort* rather than a transition for every value of the sort.) Tests would then be generated by traversing this transition graph, choosing test values on-the-fly. Unfortunately tools to achieve this are not yet available, though [1] is a promising basis. Tests for accelerators have therefore been generated by first constraining the specification behaviour. This is done by imposing input constraints with a special-purpose language that is automatically translated into LOTOS and composed with the specification.

Most work on formally-based (conformance) testing has concentrated on protocols. It has been shown that the same methods can be applied to hardware testing [12]. Protocol testing and hardware testing tend to be control-dominated. That is, the focus is on behavioural rather than data aspects. An accelerator is, however, heavily influenced by data. For example, the specification outlined in section 3 is controlled by fourteen input settings. Most of these have a very large number of possible values (e.g. dose, dose rate, positions, angles) that may be set in any order. As a result the number of possible test cases is astronomical (in excess of 10^{12}), with only a small number being interesting.

It is therefore necessary to seriously restrict the values used for testing. Fortunately, the inputs needed by an accelerator fall into two categories: values from a short list of alternatives, and numbers within defined range. An input with limited alternatives (e.g. an enumerated type) can be tested in full. An appropriate technique for ranges is boundary testing as used in software engineering. As noted earlier, numeric input parameters have been mapped to natural numbers in the specification. Suppose that some input is a natural in the range 6 to 20 inclusive. Significant test values are the lowest and highest permissible values, plus some middling value (e.g. 6, 13, 20). If it is desired to check for robustness, values just outside the permissible range should also be checked for rejection (e.g. 5, 21). It is not possible to analyse an arbitrary LOTOS data type to determine that its values lie in a bounded range. Test generation therefore relies on the specifier giving some help, namely the nature and values of bounds.

Even with these restrictions to test values, the number of test cases is still far too large because of input permutations. Instead, the specifier can help by indicating the order in which inputs may be supplied. Even if this is only a partial order, the combinatorial situation can be greatly improved.

4.2 Input Value Constraints

Only the specifier knows the intended range and ordering for input values; these cannot (reasonably) be inferred from the specification. The authors have designed PCL (Parameter Constraint Language, ‘Pickle’) to allow the specifier to give guidance on what inputs to supply and when. In general, this is a difficult problem as a wide variety of event structures and specification styles are possible with LOTOS. PCL is a reasonably general and flexible language that is applicable to many other testing problems. However it was inspired by the need to test accelerator specifications such as in section 3. See [17] for more details of PCL and its translation to LOTOS.

PCL annotations are special LOTOS comments *(*. PCL directives .*)*. As comments, they do not affect the specification behaviour. A preprocessor written by the authors extracts the PCL annotations and produces a new specification. PCL is translated into LOTOS constraint processes that are placed in parallel with the main specification behaviour. This restricts certain inputs, while leaving outputs and some inputs unconstrained. If symbolic automata are generated in future, the PCL annotations will be used to guide the generation of test values during automaton traversal.

In fact, LOTOS does not distinguish input from output so key event occurrences must be annotated. To be exact, only one example of each event structure needs to be annotated. Since the same data type may be used in different events with different sets of values, it is appropriate to annotate key events rather than data types. For example:

```
input ?prime:Nat;                                    (*. prime : values(2, 3, 5) .*)
```

This indicates that only the inputs 2, 3, 5 should be considered during test generation. (All such literals need to be defined by the specification.) The label *prime* for this set of values is used when input constraints are assembled. The label may be the same as the variable in the event, but need not be. The PCL preprocessor infers the event format from the context, e.g. *input !3* will appear in one of the test cases. In fact, an arbitrary event format can be annotated by listing sets of values in their order of occurrence:

```
input !Number ?prime:Nat !Hue ?tint:Colour ;
                     (*. prime_tint : values(2, 3, 5); values(Cyan, Magenta, Yellow) .*)
```

Here, a set of values is provided for each unbound event parameter. One test case for this event might be *input !Number !5 !Hue !Cyan*.

The **values** directive is appropriate for a limited set of discrete values. If the parameter is a number in a range (as is very common in accelerator specifications), another annotation is more appropriate:

```
read ?hour:Twelve;                                      (*. hour : range(1, 12) .*)
```

This is equivalent to the lowest, middling and highest values: 1, 6, 12. For robustness testing, the values just on each side of this range (0, 13) may be included by:

```
read ?hour:Twelve;                                     (*. hour : bounds(1, 12) .*)
```

There may be interdependencies among input values. Suppose that *size* is an input parameter defined separately. A further parameter may be defined using its value:

```
user ?value:Nat;                          (*. value : range(size + 5, 2 * size) .)
```

Since the values are computed in the specification, such expressions must be meaningful in LOTOS. Only **grouped** or **serial** inputs (see below) may be referenced in this way.

Input values may be given inside a LOTOS expression, in which case the expression is evaluated for each combination of values:

```
request ?pair:NatPair;          (*. pairs : values(MakePair(range(1, 10), values(2, 4, 8)) .*)
```

This is equivalent to **values***(MakePair(1, 2), MakePair(5, 4), MakePair(10, 8))*. Values, ranges and bounds can be nested in such a construction. The nested input lists are deliberately combined in the same order to ease predictability in testing.

Because the overall constraint process is composed with the main behaviour, it must synchronise on every event. It is therefore necessary to annotate output events and unconstrained input events so that they can be allowed to happen freely, for example:

```
file !EndOfFile ?eof:Bool;                                              (*. free .*)
```

The annotation does not have a label in this case since it is not referred to when inputs are combined. It is possible, however, to write LOTOS events such that the event format cannot be determined automatically and has to be given explicitly in the annotation:

```
choice c:Condition, b:Bool []
  [c = EndOfFile] ->
    file !c !b;                               (*. free(file !EndOfFile ?eof:Bool) .*)
```

4.3 Input Ordering Constraints

The PCL annotations so far allow the values of input events to be constrained by the specifier. To limit combinatorial explosion, it is also desirable to use further annotations to limit the possible orderings of inputs in three different ways:

separate: input values are chosen completely independently. This is the most general case, but causes the largest number of variations to be tested.

grouped: the *i* th values for inputs occur in groups, but in any relative order. Suppose that *prime* may take values 2, 3, 5 and that *tint* may take values Cyan, Magenta, Yellow. The first, second and third values from each are chosen and input in either order: 2 and Cyan, 3 and Magenta, 5 and Yellow. Grouped inputs must have an identical number of values. This is not as restrictive as it might seem since inputs are typically all defined by **range** or **bounds**. Grouping significantly reduces test combinations by checking all lowest, middling, highest, etc. values together.

serial: the *i* th values for inputs occur in sequence. Taking *prime* as first and *tint* as second, the inputs would be: 2, Cyan, 3, Magenta, 5, Yellow. This is obviously the most restrictive but least complex combination of inputs.

Each input should fall into one of these three categories (or conceivably be **free** if it is unconstrained). To limit combinatorial explosion further, **separate** or **grouped** values may be included in a **serial** list. As a further enhancement, an input may be followed by a question mark (e.g. *tax?*) to indicate an optional value. Consider a hypothetical stock control system with inputs as follows:

```
(*.
   separate(price, tax?);
   grouped(colour?, size);
   serial(weight, separate(type, code?), stock, postage?)
.*)
```

This means that inputs *price* and optional *tax* may be chosen independently. Input values for optional *colour* and *size* are chosen in groups. Serial input values are *weight*, *type* and optional *code* in either order, *stock*, and optional *postage*.

Annotations for input ordering appear after the overall LOTOS **behaviour** expression. They are extracted by the preprocessor after all the input value annotations and are used to automatically generate the constraint processes in LOTOS. This results in a new specification that is used to generate tests. Depending on the input combinations, a number of LOTOS behaviour patterns are required for the constraint processes. The accelerator test annotations in section 5 provide an illustration.

A final complication is that the specification may have cyclic behaviour. It is necessary to know when a fresh set of input values should be generated. It is assumed that some key event can be annotated as marking the end of the current cycle, e.g.:

```
output !Finished;                                          (*. finish .*)
```

5 Accelerator Test Generation

5.1 Accelerator Specification Annotations

The following input value annotations were placed in strategic places in the accelerator specification. They are scattered, but are extracted by the preprocessor:

```
mode : values(XRayMode, ElectronMode)                        (* treatment mode *)
energy : range(6, 20)                                          (* beam energy *)
dose : range(5, 100)                                            (* dose units *)
rate : range(1, 50)                                              (* dose rate *)
gantry : range(0, 359)                                        (* gantry angle *)
x1 : values(0, 0, 39)                                (* collimator X1 position *)
x2 : values(1, 40, 40)                               (* collimator X2 position *)
y1 : values(0, 0, 39)                                (* collimator Y1 position *)
y2 : values(1, 40, 40)                               (* collimator Y2 position *)
accessory : values(AccessoryIn, AccessoryOut)            (* accessory setting *)
rotation : range(0, 359)                                     (* couch rotation *)
latitude : range(0, 50)                              (* couch latitude position *)
longitude : range(0, 150)                           (* couch longitude position *)
vertical : range(60, 170)                            (* couch vertical position *)
accelerator : values(                                    (* dosimeter readings *)
  MakeStatus(values(2, 1, 2), values(2, 1, 2)),      (* first treatment readings *)
  MakeStatus(values(0, 25, 28), values(10, 26, 35)), (* second treatment readings *)
  MakeStatus(values(0, 1, 3), values(10, 50, 70)))    (* third treatment readings *)
```

Most of the input values are straightforward. The dosimeter readings are given as three lists of accelerator statuses, used on each successive treatment. Each list of *MakeStatus* values gives three pairs of dosimeter readings, chosen to stop treatment on the final value of each triple. This corresponds to the three test values for dose.

Internal (clock) events need not be marked as free since they are externally invisible. Some inputs and all outputs are marked as unconstrained. For example the console *Pause* input is unconstrained, as is the accelerator *Finish* output. In a few cases an explicit event structure must be given since it cannot be determined from the context:

```
free(console !Display ?displayNew:Display)            (* console setting output *)
free(console !Display ?interlockNew:Interlock)      (* console interlock output *)
free(accelerator !Set ?settingNew:Setting)               (* accelerator setting *)
```

Finally, the permissible combinations of inputs are given. The names refer to the input value labels given above. A number of the inputs are optional because they have default values that need not be set:

```
serial(                                                  (* sequence of inputs *)
 separate(mode?, accessory?),                    (* optional mode, accessory *)
  energy, dose, rate,                                  (* energy, dose, rate *)
   gantry?,                                         (* optional gantry angle *)
    x1, x2, y1, y2,                                   (* collimator position *)
     rotation?, latitude?, longitude?, vertical?,  (* optional couch setting *)
      accelerator)                                     (* accelerator status *)
```

5.2 Accelerator Constraint Processes

The LOTOS generated from these annotations is as follows. Different process structures result from different combinations of input constraints, so this is merely an example. Compare the annotations in section 5.1 with the following generated LOTOS.

First the overall constraint process is defined in parallel with the main behaviour. The gates of process *Constraints* are inferred from the structure of the annotated events.

Since ranges are specified for the constrained events, the lowest, middling and highest values are chosen in sequence for all the inputs. These input values are indexed 0, 1, 2 as the parameter to the *ConstraintsRepeated* process:

```
process Constraints [console, pendant, accelerator] : noexit :=      (* overall constraints *)
  ConstraintsFree [console, pendant, accelerator]                           (* free inputs *)
|||
  (
    ConstraintsRepeated [console, pendant, accelerator] (0)             (* first value set *)
  ≫
    ConstraintsRepeated [console, pendant, accelerator] (1)            (* second value set *)
  ≫
    ConstraintsRepeated [console, pendant, accelerator] (2)             (* third value set *)
  )
endproc
```

ConstraintsFree permits free events to occur without restriction:

```
process ConstraintsFree [console, pendant, accelerator] : noexit :=         (* free inputs *)
  (                                                                     (* allow one event *)
    console !Display ?displayNew:Display; exit                  (* console setting output *)
  []
    console !Display ?interlockNew:Interlock; exit            (* console interlock output *)
  []
    accelerator !Set ?settingNew:Setting; exit                   (* accelerator setting *)
  []
    ...                                                           (* other free events *)
  )
≫
  ConstraintsFree[console, pendant, accelerator]                                (* repeat *)
endproc
```

ConstraintsRepeated deals with the set of test values identified by the parameter *index*. The process allows grouped inputs to occur independently of separate and serial inputs. In this particular example, the separate events must precede the serial events. Since the specification must be tested for various cycles of behaviour, the console *Finished* event is used as the trigger to choose a new set of repeated events:

```
process ConstraintsRepeated [console, pendant, accelerator] (index:Nat) : exit :=
  (
    ConstraintsGrouped [console, pendant, accelerator]                  (* grouped inputs *)
  |||
    (
      ConstraintsSeparate [console, pendant, accelerator]              (* separate inputs *)
    ≫
      ConstraintsSerial [console, pendant, accelerator] (index)          (* serial inputs *)
    )
  )
▷
  console !Finished;                                           (* trigger to end test inputs *)
  exit
endproc
```

ConstraintsGrouped constrains nothing here, since this example has no grouped inputs:

```
process ConstraintsGrouped [console, pendant, accelerator] : exit :=     (* grouped inputs *)
  exit
endproc
```

ConstraintsSeparate deals with the separate inputs. It allows independent selection of *mode* and *accessory* inputs, each of which may be omitted as they are optional:

```
process ConstraintsSeparate [console, pendant, accelerator] : exit :=     (* separate inputs *)
  (console !Mode !XRayMode; exit [] console !Mode !ElectronMode; exit [] exit)
|||
  (console !Accessory !AccessoryIn; exit [] console !Accessory !AccessoryOut; exit [] exit)
endproc
```

ConstraintsSerial defines the serial inputs. It is also parameterised by the index of the test value set, though only set 1 is given here for brevity. The *gantry*, *rotation*, *latitude*, *longitude* and *vertical* inputs are optional and may be bypassed:

```
process ConstraintsSerial [console, pendant, accelerator] (index:Nat) : exit :=
  ...                                                        (* input value set 0 *)
[]
  (
    [index eq 1] ->                                          (* input value set 1 *)
      console !Energy !13;                                   (* middling energy *)
      console !Dose !52;                                     (* middling dose *)
      console !Rate !25;                                     (* middling dose rate *)
      console !CollimatorX1 !0;                              (* middling collimator X1 *)
      console !CollimatorX2 !40;                             (* middling collimator X2 *)
      console !CollimatorY1 !0;                              (* middling collimator Y1 *)
      console !CollimatorY2 !40;                             (* middling collimator Y2 *)
      (pendant !Gantry !179; exit [] exit)                   (* middling gantry angle *)
    ≫
      (pendant !Rotation !179; exit [] exit)                 (* middling rotation angle *)
    ≫
      (pendant !Latitude !25; exit [] exit)                  (* middling latitude *)
    ≫
      (pendant !Longitude !75; exit [] exit)                 (* middling longitude *)
    ≫
      (pendant !Vertical !115; exit [] exit)                 (* middling vertical position *)
    ≫
      accelerator !Read !MakeStatus(0, 10);                  (* middling first reading *)
      accelerator !Read !MakeStatus(25, 26);                 (* middling second reading *)
      accelerator !Read !MakeStatus(28, 35);                 (* middling third reading *)
      exit
  )
[]
  ...                                                        (* input value set 2 *)
endproc
```

As has been seen, the fairly compact annotations for input values and ordering are turned (albeit automatically) into some complex constraint processes. Through the application of PCL annotations, the accelerator specification is restricted to a manageable extent. Standard test generation techniques can then be applied.

5.3 Accelerator Test Generation

Many real-world systems communicate with their environment in a different way from an LTS. In particular, inputs and outputs are clearly distinguished. The inputs of a system are always enabled and cannot refuse the actions offered by the environment. After the system consumes an input and produces its outputs, the environment has to accept the outputs. Communication is thus no longer symmetric. In [19] this kind of behaviour is modelled as an IOLTS (Input-Output Labelled Transition System).

Several implementation relations have been defined to express conformance of an implementation to its specification. In these relations a specification is modelled as an LTS, and an implementation as an IOLTS. This is because an LTS can give a more abstract view of a system, while an IOLTS is closer to reality. The relation *ioconf* (input-output conformance) is appropriate for accelerator specifications. This relation judges an implementation to be correct if, after every trace of the specification, the implementation outputs can also be produced by the specification. An implementation cannot produce outputs that are not expected by the specification.

Checking *ioconf* can be achieved by checking trace inclusion on the suspension automaton generated from the LTS. Briefly, a suspension automaton is a directed graph built by determinising the LTS and marking quiescent states. This is indicated by adding a δ (quiescent) ‘action’ that loops back to the same state. The algorithm to transform an LTS into a suspension automaton is described elsewhere [19] and is not repeated here.

Test generation is achieved by traversing the suspension automaton. PCL annotations ensure that the accelerator specification has finite behaviour. A test suite aims to cover all transitions in the automaton. Generating a sequence to visit every edge in a graph at least once is the Chinese postman problem. As suspension automata may not be strongly connected, the approach of [6] is adopted as it is suitable for all kinds of directed graph. This method uses depth-first search whenever possible. But when an unvisited edge cannot be reached, then breadth-first search is used to find a state with an unvisited edge. The whole procedure repeats until all transitions have been covered.

Each transition tour is a test case and is saved in a test file. The test generation algorithm may find that a state offers alternative outputs with the same gate but different values. These outputs are marked when the corresponding test cases are generated, meaning they are not necessarily matched by the implementation. Execution of the test suite takes this implementation freedom into account.

CADP (Cæsar Aldébaran Development Package, [4]) supports an application programming interface that allows user-written programs to manipulate the state space of a given LOTOS specification. As reported in [12], the *TestGen* tool has been developed to generate a test suite by creating and traversing a suspension automaton. This tool was developed to generate hardware tests, but has been adapted for accelerator tests. Scalability problems mainly occur when there are multiple instances of the same processes. This has arisen with hardware (e.g. a set of bus arbiters) but not yet with accelerators.

The accelerator specification in section 3 was constrained as in sections 5.1 and 5.2. The resulting LTS, minimised with respect to observational equivalence (which respects *ioconf*), has 8616 states and 11300 transitions. There are 67 distinct paths and thus test cases. The longest has 136 events, though most are much shorter. A typical test case gives the following inputs:

mode: electron mode
energy: 13 MeV (million electron-volts)
dose: 52 MUs (monitor units)
rate: 25 MUs/minute
collimator: X1 0 cm, X2 40 cm, Y1 0 cm, Y2 40 cm
gantry: 179°
couch: rotation 179°, latitude 25 cm, longitude 75 cm, vertical 115 cm
dosimeters: 0 and 10 MUs, 25 and 26 MUs, 28 and 35 MUs

All inputs but the dosimeter ones are provided during treatment setup. Dosimeter inputs are pairs of readings during treatment. In the above, treatment stops when the accumulated dose becomes 53 MUs (0 + 25 + 28 MUs for the first dosimeter, slightly beyond the planned figure of 52 MUs). Treatment would have been aborted if the second dosimeter total had exceeded the tolerance level (52 MUs plus the fixed value 20 MUs). If the treatment time had exceeded the planned time (10% more than $\frac{52}{25}$ minutes in the above), treatment would have been aborted. Control inputs and status outputs (not included in the above) are also generated according to the accelerator specification.

6 Conclusion

Radiotherapy accelerators have been briefly described. As these are complex, software-controlled, safety-critical systems it is very desirable to systematically test their control systems. The structure of a typical accelerator specification in LOTOS has been outlined.

To have any hope of generating realistic tests, the specification must be annotated with guidance as to useful test inputs. PCL annotations define key test inputs – explicit values (say, for an enumerated type) or boundary test values (for a numeric range). Unconstrained events are also marked. Further PCL annotations define how inputs can be practicably ordered. The resulting constraint process is automatically generated and placed in parallel with the main behaviour to allow a manageable automaton to be generated. A suspension automaton is generated from this, and traversed to create test cases that form the accelerator test suite. Although PCL has been designed to help with accelerator testing, it is of general utility and should be useful for testing in other domains.

More theoretical techniques would also be interesting. For example, the constrained specifications produced by the approach lend themselves to model checking. Desirable specification properties include disallowing high-energy beams in electron mode, or forbidding certain accelerator setups. However this work is in the future. Automated test execution is being developed, so currently test cases have to be applied manually by the operator. It is also intended to investigate test generation via symbolic automata.

Although tool development is ongoing, the paper has hopefully given insight into the practicability and importance of the approach for testing radiotherapy accelerators.

Acknowledgements

This work was supported by the National Computing Centre (Manchester, *www.ncc.co.uk*). The authors are indebted to Dr. Hamish Porter (Western General Hospital, Edinburgh) for his extensive advice on accelerator design and operation. However any errors and misconceptions in the paper are due to the authors. The authors thank Dr. Jan Tretmans (University of Nijmegen) and Dr. Ji He for their insights into test generation.

References

1. M. Calder and C. E. Shankland. A symbolic semantics and bisimulation for full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. Formal Techniques for Networked and Distributed Systems*, pages 184–200. Kluwer, London, UK, Sept. 2001.
2. EC. Medical devices directive. Technical Report 93/42/EEC, European Commission, Brussels, Belgium, June 1993.
3. FDA. Medical devices: Current good manufacturing practice. Technical Report 61 FD 195, US Food and Drug Administration, New York, USA, Oct. 1996.
4. J.-C. Fernández, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDÉBARAN Development Package): A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. 8th. Conference on Computer-Aided Verification*, LNCS 1102, pages 437–440. Springer-Verlag, Berlin, Germany, Aug. 1996.
5. D. Greene and P. C. Williams. *Linear Accelerators for Radiation Therapy*. IOP Publishing Ltd., Bristol and Philadelphia, 1997.
6. R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proc. 22nd. Annual International Symposium on Computer Architecture*, 1995.
7. IEC. *Medical Electrical Equipment – Part 2: Particular Requirements for Safety*. IEC 601-2. International Electrotechnical Commission, Geneva, Switzerland, 1988.
8. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.
9. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Enhanced LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 15437. International Organization for Standardization, Geneva, Switzerland, 2001.
10. J. Jacky, J. Unger, M. Patrick, D. Reid, and R. Risler. Experience with Z developing a control program for a radiation therapy machine. In J. P. Bowen, editor, *Proc. 10th. International Conference of Z Users*, LNCS. Springer-Verlag, Berlin, Germany, Dec. 1996.
11. J. Jacobson and O. Andersen. Software controlled medical devices. Technical Report SP-Rapport 1997:11, European Network of Clubs for Reliability and Safety of Software, Apr. 1997. ISBN 91-7848-669-6.
12. Ji He and K. J. Turner. Protocol-inspired hardware testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Proc. Testing Communicating Systems XII*, pages 131–147. Kluwer, London, UK, Sept. 1999.
13. E. J. Joyce. Accelerator linked to fifth radiation overdose. *American Medical News*, 1, Feb. 1987.
14. C. J. Karzmark. Procedural and operator error aspects of radiation accidents in radiotherapy. *International Journal of Radiation Oncology Biological Physics*, 13:1599–1602, Jan. 1987.
15. N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
16. R. Nath, P. J. Biggs, F. J. Bova, C. C. Ling, J. A. Purdy, J. van de Geijn, and M. S. Weinhous. AAPM code of practice for radiotherapy accelerators. *Medical Physics*, 21(7):1093–1121, July 1994.
17. Qian Bing and K. J. Turner. Input value constraints for radiotherapy accelerator specification. Technical Report CSM-161, Computing Science and Mathematics, University of Stirling, UK, Aug. 2002.
18. M. H. Thomas. The story of the Therac-25 in LOTOS. *High Integrity Systems Journal*, 1(1):3–15, Feb. 1994.
19. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.