Kenneth J. Turner. Test Generation for Radiotherapy Accelerators,
Software Toolsfor Technology Transfer, 7(4):361-375, Springer, August 2005

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Test Generation for Radiotherapy Accelerators

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Scotland FK9 4LA
e-mail: kjt@cs.stir.ac.uk

September 7, 2005

Abstract System specification with LOTOS (Language Of Temporal Ordering Specification) is briefly introduced. To make test generation practicable, specifications are annotated with event constraints using PCL (Parameter Constraint Language) as a means of stating test purposes. Automated test generation can then use the principle of input-output conformance to check whether an implementation agrees with its specification. Test suites are generated by a transition tour that either visits every transition at least once (for infinite behaviour) or follows every path (for finite behaviour). The approach is applied to a case study in which tests are generated for radiotherapy accelerators used in cancer treatment. A typical specification and set of test purposes yields 256 test cases that can be executed manually or automatically. The goal is to determine situations in which an accelerator does not behave in conformity with its specification.

Key words Accelerator – LOTOS (Language Of Temporal Ordering Specification) – Radiotherapy – Test Generation

1 Introduction

1.1 Background

In general, testing is a time-consuming and exacting task. Testing can very rarely be exhaustive, so there is an issue of effective coverage. In practice, tests are often defined using human expertise and are conducted manually. Test procedures differ greatly according to the domain. In electronics, for example, a sharp distinction is made between testing for *design* errors and testing for *manufacturing* errors. The former is often termed design verification. Software testing almost entirely focuses on design flaws. The techniques employed include black/white box testing, alpha/beta testing, module/system testing, and integration/regression testing.

In this paper, the focus is on conformance testing – a concept that arises with communications protocols. The goal is to check the agreement of an implementation with its defining specification. Conformance testing is particularly well developed for communications protocols, where a methodology and framework have been standardised [22, 25].

Techniques have been developed for the use of formal methods in conformance testing [23]. Ideally, a system would be rigorously developed using a formal method throughout the whole of its development. The conformance of an implementation to its specification (i.e. formal requirements) would then be inevitable. In practice, this does not apply for a number of reasons:

- It is rare for a formal method to be sufficiently wide-spectrum that it can address development at all levels, from requirements to implementation.
- In general, industrial engineers have little training in formal methods. Formal methods tend to be used only in special kinds of systems (e.g. quality-critical or safety-critical ones).
- Stepwise refinement of a specification towards an implementation takes considerable effort. Even if it is done, there is almost inevitably a gap at the stage of final implementation. For example, it is very unusual for the compiler that generates executable code to have been verified. Equally, the operating system that runs the code is most unlikely to have been verified.
- Progressing from specification to implementation, the size and complexity of the code increases greatly. Present day verification techniques can only just cope with realistic specifications. Verification of realistic implementations is impracticable.

As a result, it is almost always necessary to check that an implementation is indeed conformant.

However, formal methods still have an important role to play in testing. Given a specification, it is possible to generate useful tests from it. This is not, of course, as complete as proving that an implementation respects its specification. However it can be used to gain confidence in the implementation. Furthermore, a high degree of automation can be used to generate and apply the tests.

1.2 Radiotherapy Accelerators

Radiotherapy equipment is used in oncology (cancer) centres to deliver controlled doses of radiation, usually to destroy cancerous tissue. Among the several kinds of radiotherapy equipment, the most important is the linear accelerator ('accelerator' or 'linac'). This accelerates a beam of electrons to high energy that can be used for treatment directly or to generate x-rays.

Radiotherapy is a safety-critical procedure that demands accurate delivery of radiation. A number of radiation accidents have been well documented (e.g. [34, 35]). The Therac-25 accelerator is infamous as having caused accidental injuries, in some cases leading to death [37]. 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.

The accelerator is located in a treatment room that 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. [16] gives a comprehensive introduction to the theory and practice of treatment with radiotherapy accelerators.

Radiotherapy equipment uses dedicated hardware. The physical characteristics 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 deal with situations like power supply failure, dosimeter failure, or entry to the treatment room while the accelerator is operating.

Early radiotherapy equipment was essentially just hardware. However, modern accelerators are complex software-controlled systems. 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.

1.3 Conformance Testing

The aim of the work reported in this paper was to adapt protocol techniques to testing the control system of radiotherapy accelerators. At first sight, this might seem to be an implausible approach. However, the author was encouraged by good experience of using protocol techniques for hardware testing [31]. This gave some confidence that the same approach could be useful in testing accelerators, and in fact for testing medical equipment more generally.

For protocols, it has been found useful to employ an implementation relation called *ioconf* (input-output conformance). The basic idea is to formally relate the input-output behaviour of a specification to that of its implementation. The approach distinguishes implementation states where output is not possible, i.e. the implementation is awaiting further input. All sequences of behaviour (traces) are considered for the specification. An implementation conforms to its specification if the outputs of the implementation after such traces can also be produced by the specification.

Although *ioconf* was developed to evaluate communications protocols, it has proven possible to adapt it for testing accelerators. The formal basis is a specification language borrowed from the communications world. LOTOS (Language

Of Temporal Ordering Specification [21]) was originally intended for specifying communications systems, but has subsequently been used for many other kinds of systems.

1.4 Methodology

The methodology used in this paper is generic, and could be applied to test generation for other kinds of systems. However the paper grounds the approach in a particular application domain: radiotherapy accelerators. The steps in the method are as follows. In each case, a brief description of the step is followed by what was actually done by the author (and co-workers) in the accelerator case study. Subsequent sections of the paper elaborate on each of these steps.

Information gathering: an understanding of the application is gathered from discussions with domain experts.

The author collaborated with a radiation physicist who is responsible for operation of radiotherapy accelerators in an oncology centre. This allowed the author to gain a technical understanding of accelerator control systems. Discussions were also held with a major accelerator manufacturer, but it proved to be impracticable to involve them in the research. Technical information was therefore based on knowledge gleaned from the collaborating oncology centre.

Modelling: an informal model of the system is created.

Structural breakdowns and data-flow diagrams were produced for a sample accelerator. Clarifications were sought from domain experts throughout this process, though the technical representations that emerged were the author's. An issue at this stage was choosing an appropriate level of abstraction. The initial functional breakdown led to a rather detailed model. However most of this did not deal with the core functionality. Attention was therefore focused on the control system of a radiotherapy accelerator as a black box. The input-output behaviour of this characterises the important aspects of the control system.

Specification: a formal specification is written based on the informal model.

LOTOS was used as the specification language. The specification reflects both the structural breakdown of the system as well as insights gained during modelling its functional behaviour. As far as the author knows, (formal) specifications are not written of accelerators so this was a necessary and useful step.

Constraint annotation: to make test generation feasible, the specification is annotated to indicate significant test values and orderings.

PCL (Parameter Constraint Language) was devised by the author to guide test generation in a useful and practical manner. PCL allows the specifier to define which kinds of test values are important, as well as to define which orderings of inputs are significant. Test constraints were formulated for accelerators, partly following general software engineering practice and partly using

domain knowledge. The specification was annotated with the corresponding test constraints. The PCL tool automatically translates these to LOTOS and combines them with the original specification. The resulting specification is thus restricted to behave within useful test constraints.

Automaton generation: a finite state automaton is automatically generated from the annotated specification, using the test constraints to make the state space manageable.

The automaton was created automatically by a standard toolset for LOTOS – CADP (Cæsar Aldébaran Development Package). The automaton was minimised (with respect to observational equivalence) in order to make test generation more efficient.

Test generation: a suite of tests is automatically generated by traversing the automaton, following an existing algorithm that respects the *ioconf* relation.

A test suite was created by traversing the automaton, using an adaptation of the *ioconf* algorithm. The resulting tests are known to be sound and consistent. The TestGen tool was developed for this purpose, though similar tools (TGV, TorX) have been subsequently created by others.

Test application: the suite of tests is automatically run against the implementation, with a view to deciding whether it conforms to its specification.

This is the point which the research has reached. Currently the tests can be applied only manually, though a strategy has been devised for running them automatically. The goal is to run the test suite periodically to confirm satisfactory operation of the accelerator – particularly after a software upgrade.

1.5 Related Work

Formal methods are an obvious choice to support the development and testing of radiotherapy equipment. A big impetus to the use of formal methods was given by a series of accidents involving the Therac-25 accelerator [38]. However rather surprisingly, radiotherapy equipment continued to attract little attention from the formal methods community. [47] is one of few contributions, having made use of LOTOS to show (with the benefit of hindsight) how the Therac-25 flaws could have been identified. The only other work known to the author uses Z to specify the design of software for a radiation therapy machine [26–29].

In formally-based conformance testing, a specification of the target system is presumed to exist. Typically this is represented by an LTS (Labelled Transition System) that can give the semantics of a behaviourally-oriented specification language like LOTOS. Test theories for LTSs have been under investigation for some time, based on external tests and observations (e.g. [8,43]). The theories support implementation relations that formally qualify an implementation with respect to its specification. Apart from defining a suitable implementation relation, conformance testing requires finding a set of tests for a specification to distinguish between correct and incorrect implementations.

[3] elaborates a theory for testing systems specified in LOTOS. Several test generation algorithms have been proposed for an LTS corresponding to Basic LOTOS (i.e. LOTOS without data types), e.g. [36,44]. In [48,49] the testing theory for an LTS is refined for communicating systems that distinguish inputs and outputs.

A formal description of an implementation rarely exists, either because the implementation is opaque or because it would be impracticable to specify it. However by what is known as a test hypothesis, it is presumed that the implementation can be modelled as an IOLTS (Input-Output Labelled Transition System). An LTS gives a relatively abstract description (and so is appropriate for a specification), while an IOLTS gives a more realistic and concrete description (and so is appropriate for an implementation).

Conformance of an implementation can be expressed with respect to its specification using a formal relation between the IOLTS and LTS. The *ioconf* relation [49] can be used as a criterion for correct implementation. Based on this relation, an algorithm has been given for defining a suite of implementation tests [49]. A test suite consists of test cases that define possible inputs and expected outputs. Another article in this special issue [30] provides the theory and principles behind conformance-based test generation. The present article therefore gives a less technical treatment of *ioconf*.

The author and his co-workers have implemented this algorithm by building on the API for the CADP toolset (Cæsar Aldébaran Development Package [12]). Originally the goal was to generate tests for hardware [31–33]. Subsequently the approach has been modified for testing radiotherapy accelerators.

The approach is similar to that of the test generation tool TGV [13]. However because a radiotherapy accelerator specification is heavily dependent on data, TGV is not immediately useful. TGV also requires an accurate knowledge of the state space of a specification, which is not known until a specification has been constrained for testing. TorX is a similar tool for LOTOS-based test generation, but was not available to the author as it was developed for use in a specific project. STG (Symbolic Test Generation [7]) could be particularly relevant to accelerator test generation.

For hardware testing, formal methods have been combined with simulation techniques. In [54], software testing methods are used for design verification of behavioural VHDL (VHSIC Hardware Design Language [19]). In [17,42] test generation is based on an FSM (Finite State Machine) or ECFM (Extracted Control Flow Machine) that represents the control logic of a circuit. The generated test cases are then applied to both higher level and lower level specifications in Verilog [20] or VHDL. These approaches are built on a formal model *extracted* from a circuit design. However the author favours an approach in which tests are *derived* from a high-level specification. [46] generates tests from a higher-level FSM specification, and applies them using a VHDL simulator. Unfortunately this method cannot handle non-determinism in specifications.

1.6 Overview of Paper

Section 2 introduces system specification with LOTOS. To make test generation practicable, it is necessary to constrain the specification using PCL (Parameter Constraint Language) as a means of stating test purposes. Test generation using input-output conformance is explained in section 3. A test suite can be automatically generated according to various strategies. The main case study is presented in section 4, where radiotherapy accelerators are described. Sample test annotations for accelerators are given, and the resulting test suite is discussed. The paper concludes with an evaluation of the approach in section 5.

2 Specification for Testing

2.1 Specification with LOTOS

LOTOS (Language Of Temporal Ordering Specification [21]) was originally conceived for specifying communications systems. However it is a general-purpose language that has been used in other domains. For example[1] it has been used to specify bus architectures [5], computer-integrated manufacturing [39], embedded systems [6], graphics [45], hardware design [57], multimedia systems [1], neural networks [15], object-oriented software [41], telephony [11], transaction processing [56], user interfaces, visualisation [53], and voice services [51].

LOTOS specifies behaviour using a process algebra based on CCS (Calculus of Communicating Systems [40]) and CSP (Communicating Sequential Processes [18]). Abstract data type specification is based on ACT ONE [10]. Although LOTOS is a constructive specification language, it is possible to use it for fairly high-level descriptions of systems [55].

Among languages that might be used to specify radiotherapy accelerators, LOTOS is a good choice for the following reasons:

- Its constructive nature is appropriate for giving behavioural descriptions.
- LOTOS ties in well with theories for test generation.
- LOTOS is well supported by readily available tools.

The subset of the LOTOS notation appearing in this paper is summarised below. In-line comments are also given to explain specification constructs as they are used. Tutorials on LOTOS can be found in [2,50].

Data Types: A LOTOS data type such as *NaturalNumber* (non-negative integers) has a sort (i.e. type, *Nat* in this case) and operators (e.g. '+'). The LOTOS library offers standard data types, and others can be defined by the specifier.

Actions: A behaviour finishes (deadlocks) with **Stop**. A behaviour is considered to finish successfully with **Exit**. Actions are events that occur at gates, which act like ports where communication may occur. A fixed event parameter has the form '!*value*', and is often used to output a value. A variable event parameter

[1] The citations here are representative samples from a much larger list.

has a form like '?*variable*:*sort*', and is often used to input a value; a value of the given sort is assigned to the variable. It is possible to mix several '!' and '?' parameters in an event.

Processes: A process encapsulates parameterised behaviour in the form:

Process *process* [*gates*] (*parameters*) : *result* :=
...
EndProc

When the process is called, specific gates and parameter values are provided. The result may be **Exit** (if the behaviour exits) or **NoExit** (if the behaviour stops or repeats indefinitely).

Operators:

B1 ≫ *B2* ('enables'): continues with *B2* if *B1* exits.

B1 ||| *B2* ('interleaves'): allows the events of *B1* and *B2* to occur independently in parallel.

B1 || *B2* ('synchronises'): requires *B1* and *B2* to agree on all events.

Choice *variables* [] *B*: allows *B* for all possible combinations of variable values (as defined by their sorts).

2.2 Constraining LOTOS Specifications

2.2.1 Constraint Annotations Once a LOTOS specification has been written, various analyses can be performed:

- The specification can be animated or simulated manually to check its behaviour.
- The state space of the specification can be explored to detect deadlocks, livelocks, unreachable states and unspecified receptions.
- Desirable properties of the service can be formulated in a temporal logic (e.g. ACTL or XTL) and model-checked against the specification.

However, even for small specifications this can be very time-consuming or impracticable. A more pragmatic use for a specification is generating tests from it. Assuming the specification is a faithful reflection of the intended behaviour, automated test generation can be used to gain confidence in the implementation. However direct test generation is typically impractical – especially if the specification makes extensive use of data.

In protocol testing, it is common to restrict the behaviour of a specification by imposing test purposes that constrain the behaviour to be tested. For example a test purpose might check what happens between sending data and its reception. A comparable approach has been adopted for testing radiotherapy accelerators, but the focus is on the selection of data values since this kind of specification is heavily data-oriented.

PCL (Parameter Constraint Language or 'Pickle') was developed by the author as a means of guiding test generation through test purposes. PCL annotations are attached to important parts of the specification. Only the specifier knows the plausible values and ordering of inputs; these cannot (reasonably) be inferred from the

specification. PCL adapts what is called boundary value testing in software development. If values in a range must be accepted, it is worthwhile checking just inside and just outside the range.

An extra complication is that concurrency in a specification may allow inputs to be provided in many different orders. PCL defines constraints on event values in isolation, and on the order of events. Normally PCL is used to restrict only input events, but it can also be applied to outputs (e.g. to limit the responses from a system). If the constraints are tighter, fewer variations have to be tested but the tests become less comprehensive.

PCL takes the form of special LOTOS comments *(*. PCL .*)*. As comments, these do not affect the formal meaning of the specification so normal analytic techniques apply. However the PCL translator tool can turn such annotations into LOTOS constraints that restrict the specification. Two approaches could be adopted for test constraints:

- Ideally, a symbolic transition system would first be created from the specification. Transitions would give event variables as names rather than as specific values of their sorts. Tests could then be generated by traversing this symbolic transition graph, choosing test values according to the PCL constraints. [7] describes a symbolic test generation tool that could be useful. [4] is also a promising basis. Testing of algebraic data types and processes is discussed in [14]. Symbolic execution of LOTOS specifications is well established (e.g. [9]).
- More practically, the PCL constraints can be applied immediately to the specification. This reduces its state space to a manageable size so that standard test generation algorithms can be applied.

PCL is translated automatically into LOTOS. In fact, the constraints could be written directly in LOTOS. However, as will be seen the constraints are rather complex when expressed in LOTOS. PCL is a much more compact notation that links test purposes closely with system behaviour. It is therefore preferable to use PCL and to have the LOTOS constraints generated automatically.

2.2.2 Event Value Constraints Table 1 summarises the PCL annotations for constraining event values. The ***values*** constraint ensures that tests are generated only for specific values that are thought to be useful. For values within a specific numerical range, ***range*** is used. The environment may also be allowed to provide out-of-range values with ***bounds***.

An event may be followed by a PCL value constraint. A constraint may be labelled for use in other constraints. Event parameters are fixed values ('!' prefix) or variables ('?' prefix). One constraint is given for each variable value.

Suppose the *check* event can accept a *lower* value in the range 4 to 10, and a *higher* value with useful test values 2, 5 and 6. The value *mid* in this event is fixed. The *mixture* constraint might appear as:

```
check ?lower:Nat !mid ?higher:Nat          (* check event for low/mid/high *)
  (*. mixture : range(4,10); values(2,5,6) .*)
```

Constraint	Meaning
bounds(*low*,*high*)	like ***range***, but also including *low*-1 and *high*+1 for robustness testing
free(*event*)	no value restrictions
range(*low*,*high*)	a continuous numerical range, with exemplar test values *low*, $\lfloor \frac{low+high}{2} \rfloor$ and *high*
values(*value1*,*value2*,...)	a list of specific values that may be chosen

Table 1 PCL Value Constraints

Each constraint must have the same number of alternative values: three in this case. These are chosen in tandem, so the pairs of test values are (4,2), (7,5) and (10,6).

If a constraint defines a single list of values, it may be used symbolically in another constraint. Suppose the width that is input for a rectangle should be in the range 2 to 20 (i.e. test values 2, 11, 20). The height that is also input might then be restricted to a range 6 to 12 more than the width. These constraints are expressed as follows:

```
rectangle ?w:Nat                          (* rectangle event for width *)
  (*. width : range(2,20) .*)
rectangle ?h:Nat                          (* rectangle event for height *)
  (*. height : range(width+6,width+12) .*)
```

If the test value for *width* is 11, for example, the *height* would be selected from the range 17 to 23 (i.e. test values 17, 20, 23).

LOTOS operations may be used in PCL constraints. If test values are given as operation parameters, the operation is applied to these. In such a case, constraints are often nested. Suppose the *MakeStatus* operation takes a pair of numerical values. The expression *MakeStatus*(***values***(0,25,28),***values***(10,26,35)) applies the operation to the corresponding pairs of values: *MakeStatus*(0,10), *MakeStatus*(25,26) and *MakeStatus*(28,35). In the following example, the outer call of ***values*** offers three such lists of values, i.e. nine *MakeStatus* values in total:

```
accelerator !Read ?status:Status          (* accelerator event to read status *)
  (*. accelerator : values(
        MakeStatus(values(2,1,2),values(2,1,2)),
        MakeStatus(values(0,25,28),values(10,26,35)),
        MakeStatus(values(0,1,3),values(10,50,70))) .*)
```

If an event has no PCL constraint, its values are unrestricted. Since test generation makes a distinction between input and output events, it is necessary to annotate an unconstrained input event as ***free***. The PCL translator can normally infer the structure of an event, but in this case the structure might be impossible to determine. Consider the following file action in which only the second parameter may vary, and that in an unconstrained way. It would be difficult to determine that *result* was fixed, so the underlying event structure is made explicit in the constraints.

```
Choice result:Condition, buffered:Bool []   (* for all value combinations *)
  [result = OK] ->                                    (* result is OK? *)
    read !result !buffered;                           (* read buffered value *)
```

Constraint	Meaning
alternate(*label1*,*label2*,...)	the *i*th values are selected as alternatives
finish	the event ends a cycle of behaviour
grouped(*label1*,*label2*,...)	the *i*th values are selected in either order
separate(*label1*,*label2*,...)	the constraints are applied independently
serial(*label1*,*label2*,...)	the *i*th values are selected in sequence

Table 2 PCL Ordering Constraints

(*. ***free***(read !OK ?buffered:Bool) .*)

2.2.3 Event Order Constraints Although value constraints significantly restrict what must be tested, concurrency in the specification may allow impracticably many variations in the order of events. For example there are many parameters to be set before radiotherapy accelerator treatment begins, but the ordering of these inputs is largely irrelevant. Testing all the orders *could* be significant, but would probably not be. As summarised in table 2, PCL allows event ordering constraints to be defined for lists of labelled value constraints. In the examples below, suppose the following value constraints have been defined: *device* provides the test values *keyboard*, *mouse*, *pen*; and *resolution* provides the test values 0, 10, 20.

- The ***separate*** constraint allows any order of inputs. Twenty interleaved combinations would be defined by ***separate***(*device*,*resolution*).
- To limit the combinations, ***grouped*** can be used to select a value from each list in combination (each list having the same number of values). Eight combinations would be defined by ***grouped***(*device*,*resolution*): *keyboard* and 0 in either order, then *mouse* and 10 in either order, then *pen* and 20 in either order.
- The values of each group can be chosen as alternatives. (For two lists of values, this does not reduce the number of combinations but does reduce the number of inputs.) Eight combinations would be defined by ***alternate***(*device*,*resolution*): *keyboard* or 0, then *mouse* or 10, then *pen* or 20.
- The most restrictive combination is serial: the first value in each list is chosen, then the second value, etc. Again, there must be the same number of values in each list. Just one combination would be defined by ***serial***(*device*,*resolution*): the sequence of inputs *keyboard*, 0, *mouse*, 10, *pen*, 20.

Further variants of these combinations are possible. An individual list of values may be made optional by following it with a question mark: '*device*?' means this input may or may not occur. An entire combination may also be made optional: ***alternate***?(*device*,*resolution*). Ordering constraints may be given individually or may be nested. All ordering constraints are stated after the main LOTOS behaviour expression. The following is drawn from the radiotherapy accelerator test annotations:

Behaviour (* overall behaviour *)
Accelerator [Console] (* accelerator behaviour *)
(*.

```
        serial(                                          (* values in sequence *)
          separate(mode),                              (* mode values separately *)
          separate(accessory),                   (* accessory values separately *)
          serial(                                        (* values in sequence *)
            energy, dose, rate,                    (* energy/dose/rate values *
            x1, x2, y1, y2,                          (* x1/x2/y1/y2 values *)
            alternate?(                             (* one of each optionally *)
              gantry                                        (* gantry values *)
              rotation, latitude, longitude, vertical       (* couch values *)
            ),
            start, accelerator                 (* start point, accelerator values *)
          )
        )
      .*)
```

The above defines the order in which certain values may be provided during test generation. It makes use of value constraints like *mode* and *accessory* that are defined as explained in section 2.2.2. The specific value constraints used here will be given in section 4.3 once the domain of radiotherapy accelerators has been introduced..

A specification is often cyclic: it accepts input, produces some output, and then repeats this behaviour. In such a case, a PCL annotation is given to indicate the event that marks the end of a cycle:

```
console !Done;
  (*.finish .*)
```

2.3 Translating Event Constraints to LOTOS

The PCL translator extracts annotations from a specification and translates them into LOTOS constraint processes. The principles of the translation are discussed in [52] and are not given here.

At the end of section 2.2.3, ordering constraints were given for the example of a radiotherapy accelerator. These are translated in the following LOTOS process structure that reflects the constraints:

```
Behaviour                                            (* overall behaviour *)
  Accelerator [Console]                            (* accelerator behaviour *)
||                                                   (* synchronised with *)
  Constraints [Console]                            (* constraint behaviour *)
Where                                                (* local definitions *)
  Process Constraints [Console] : NoExit :=          (* overall constraints *)
    ConstraintsFree [Console]                      (* free event constraints *)
  |||                                                (* interleaved with *)
    (
      ConstraintsSerial1 [Console]                 (* top serial constraints *)
    >>                                                   (* followed by *)
      Stop                                               (* finish of tests *)
```

```
  )
EndProc
Process ConstraintsFree [Console] : NoExit :=    (* free event constraints *)
  ...                                            (* individual free events *)
 ≫                                                        (* followed by *)
  ConstraintsFree [Console]                      (* repeat free constraints *)
EndProc
Process ConstraintsSerial1 [Console] : Exit :=    (* top serial constraints *)
  ConstraintsSeparate1 [Console]             (* separate mode constraints *)
≫                                                        (* followed by *)
  ConstraintsSeparate3 [Console]          (* separate accessory constraints *)
≫                                                        (* followed by *)
  ConstraintsSerial5 [Console] (0)              (* first serial constraints *)
≫                                                        (* followed by *)
  ConstraintsSerial5 [Console] (1)             (* second serial constraints *)
≫                                                        (* followed by *)
  ConstraintsSerial5 [Console] (2)              (* third serial constraints *)
EndProc
...
```

The constraint processes are all automatically generated from the PCL. *Constraints* defines all the constraints, synchronised with the main accelerator behaviour. *ConstraintsFree* deals with free events, interleaved with *ConstraintsSerial1* for the top-level serial combination. The latter (and in fact the whole specification) terminates once all test combinations have been chosen. *ConstraintsFree* allows a free event to occur and then repeats.

ConstraintsSerial1 defines the top-level serial constraints. *ConstraintsSeparate1* gives the *mode* constraints, while *ConstraintsSeparate3* gives the *accessory* constraints. Then the remaining serial constraints are given by *ConstraintsSerial5*. This provides three lists of test values, indexed as 0, 1, 2 in the LOTOS translation. These values specify *energy*, *dose*, *rate* and *x/y* values. At this point, *ConstraintsAlternate6* (not shown) optionally allows for alternative values of *gantry*, *rotation*, *latitude*, *longitude* and *vertical* settings. Finally, it applies the *start* and *accelerator* constraints.

3 Test Generation

3.1 Input-Output Conformance

See [30] in this special issue for the theory behind the approach described here. A specification is assumed to be modelled by an LTS (Labelled Transition System) that can be generated from, say, a LOTOS specification. In early work on theories for conformance testing, both the specification and the IUT (Implementation Under Test) were modelled by LTSs. To formally define the relationship between an implementation and its specification, a test hypothesis is needed that the implementation can be represented by a formal model. The IUT communicates with

its environment through symmetric interactions, so the test environment is also modelled as an LTS.

However in many real-world systems, there is a clear distinction between input and outputs. The inputs of a system are always enabled and cannot refuse the actions offered by the environment. After the system consumes an input, the environment must be prepared to accept the resulting output. In [49] this kind of behaviour is modelled as an IOLTS (Input-Output Labelled Transition System). This is an LTS in which the set of actions is strictly partitioned into inputs and outputs. Quiescent states in an IOLTS are ones where only input is expected, i.e. output is not permitted. Such states are labelled with the δ pseudo-action that means the systems idles while waiting for input.

The specification LTS can be regarded as a partially specified IOLTS in the sense that there are some states in the specification that can refuse input actions. This may be because it does not matter how implementations respond to unexpected inputs, or because the environment should not offer them anyway.

The goal is to show that an implementation is input-output conformant with respect to its specification, i.e. that it respects the *ioconf* relation. After all traces of the specification, the outputs of the implementation must also be possible for the specification. Since this holds also for δ actions, the implementation may not output if the specification cannot do so.

Test cases respecting *ioconf* are generated from an intermediate LTS called a suspension automaton that is built from the specification LTS. The suspension automaton is obtained by adding δ self-loops for all quiescent states, and then making the resulting automaton deterministic. Checking *ioconf* then amounts to checking that implementation traces are included in those of the suspension automaton.

A test case is a finite, deterministic LTS with *Pass* and *Fail* states. A test suite is a set of such test cases. For accelerator testing, a slightly modified form of the algorithm in [49] is used. The following alternative choices are repeatedly made during test case generation:

Choice 1: The test case is terminated with a *Pass* verdict. Since a specification may have infinite behaviour, test generation must be stopped at some point – hopefully after adequate test coverage has been obtained.

Choice 2: An input is selected from the traces of the suspension automaton. This is fed to the implementation, and the algorithm repeats to make further choices. Since inputs are always enabled, no deadlock can occur. To avoid unnecessary non-determinism during testing, only one input is applied at a time.

Choice 3: Check the outputs of the implementation against the specification. If the implementation can output something that is forbidden by the suspension automaton of the specification, a *Fail* verdict is given. Otherwise the algorithm repeats.

3.2 Test Case Example

To illustrate test generation, it is simpler to use a hardware example [31]. A hardware specification needs only simple data types (bits), whereas the accelerator specification is much more complex and uses many data types.

Consider a basic logic design element: a JK flip-flop. This is a single-bit memory with control inputs *J* and *K*. If they are both set to 0, the flip-flop state stays the same. If they are both set to 1, the flip-flop inverts its stored value. If *J* and *K* are set to different values, the value of *J* is stored. The output is conventionally called *Q*, while its complement is *NQ* (not *Q*). It can be specified by a LOTOS process as follows. The parameter *data* is set to 0 when the process is instantiated.

```
Process JK [J,K,Q,NQ] (data:Bit) : NoExit :=                (* JK flip-flop *)
  J ?newJ:Bit;                                              (* get new J value *)
  K ?newK:Bit;                                              (* get new K value *)
  (
    [(newJ Eq 0) And (newK Eq 0)] ->                        (* J and K both 0? *)
      Q !data;                                              (* output current data *)
      NQ !Not(data);                                        (* output inverted data *)
      JK [J,K,Q,NQ] (data)                                  (* repeat for same state *)
  []
    [(newJ Eq 1) And (newK Eq 1)] ->                        (* J and K both 1? *)
      Q !Not(data);                                         (* output inverted data *)
      NQ !data;                                             (* output current data *)
      JK [J,K,Q,NQ] (Not(data))                             (* repeat for opposite state *)
  []
    [newJ Ne newK] ->                                       (* J and K differ? *)
    Q !newJ;                                                (* output J value *)
    NQ !Not(newJ);                                          (* output inverted J value *)
    JK [J,K,Q,NQ] (newJ)                                    (* repeat for J value *)
  )
EndProc
```

The left-hand diagram of Fig. 1 shows a minimised LTS generated from this specification. The right-hand diagram shows the corresponding suspension LTS. Since the specification is deterministic, the suspension automaton requires only δ self-loops where further input is expected. In general, the suspension automaton differs significantly where non-determinism has to be unfolded.

Fig. 2 shows sample test cases generated by traversing the suspension automaton of Fig. 1. For convenience, test cases are grouped in the diagrams where they share a common prefix. Each test case is a single sequence that starts with *J !1* and finishes at a leaf node of the diagram. Fig. 2 thus illustrates ten separate test cases.

3.3 TestGen Tool

The principal author of [31] developed an initial version of TestGen tool, embodying the algorithm in section 3.1. This made use of the API for the CADP toolset (Cæsar Aldébaran Development Package [12]).

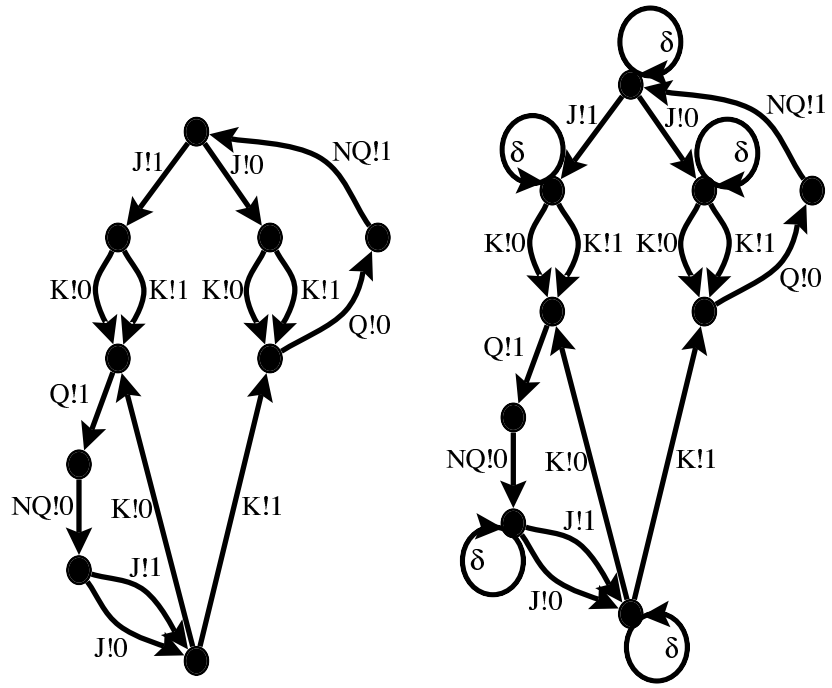

Fig. 1 Specification LTS and Suspension LTS for JK Flip-Flop

For the work reported in this article, a more elaborate version of the TestGen tool was developed and coupled with use of PCL. Although test generation is automated (with a *makefile*), quite a number of stages are involved:

- The PCL annotations in the specification are translated to LOTOS and combined with the original to make a new specification.
- A header file in C is generated for the LOTOS data types. An LTS is then generated in Aldébaran format for the LOTOS behaviour. This is minimised with respect to observational equivalence (which respects the *ioconf* relation).
- A header file and a code file in C are created for the minimised specification. All the code is then compiled and run to generate the tests.

TestGen needs to classify events as inputs or outputs. This is achieved by a separate file in C that recognises output events using regular expressions. An event pattern may refer simply to the event gate or to any parts of the event. For the radiotherapy accelerator, for example, all events at the *Couch* gate are outputs, while only events with a *Display* or *Finished* parameter are outputs for the *Console* gate.

A test suite aims to cover all transitions in the suspension automaton. Note that this is not the same as following paths through the LOTOS source, since the suspension automaton is based on a minimised and more abstract representation of behaviour. For a specification with infinite behaviour, TestGen can perform an edge tour of the suspension automaton. Visiting every edge in a graph at least once

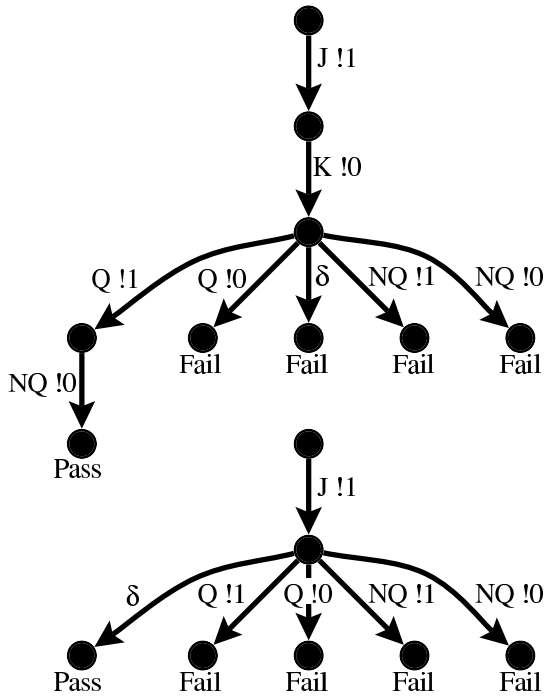

Fig. 2 Some Test Cases for the JK Flip-Flop

is the Chinese postman problem. As suspension automata may not be strongly connected, the algorithm given by [17] was adapted 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.

For a specification with finite behaviour, TestGen can perform a complete traversal of the suspension automaton (up to some specified limit on the number of tests). If the specification has been restricted by PCL constraints, this will ensure that the specification always terminates. When PCL is used with radiotherapy accelerator specifications, this kind of transition tour is appropriate for test generation.

4 Case Study

4.1 Radiotherapy Accelerators

A typical radiotherapy accelerator is shown schematically in Fig. 3. 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

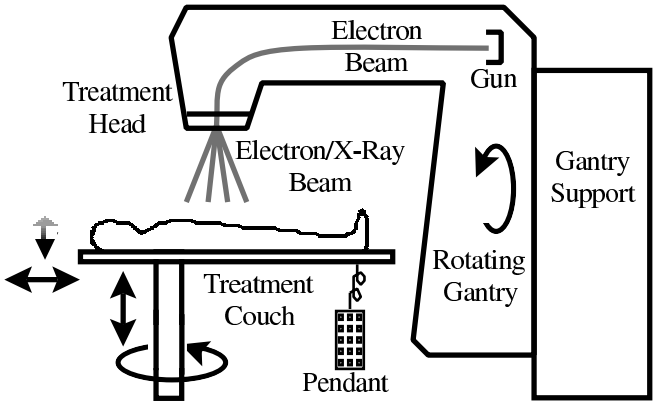

Fig. 3 Accelerator Outline

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' may also 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.

4.2 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

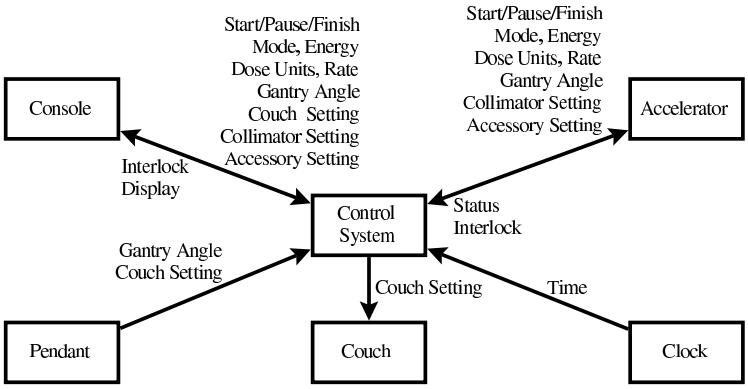

Fig. 4 Simplified Accelerator Control System

the second one are used as a backstop. 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 a specified percentage.

For the work reported in this paper, the control system has been simplified as shown in Fig. 4. The detailed information flows are shown against the arrows. All control functions are thus grouped in a single black box, with the main inputs and outputs as shown. Although the real system involves considerable communication among subsystems, Fig. 4 is a legitimate abstraction since it shows only the externally observable interfaces. The LOTOS specification reflects this black-box view of the control system.

More details of the control system and its LOTOS specification appear in [52]. 730 lines of LOTOS are required to specify a typical accelerator, about half of these dealing with data types. 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 [24]) 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 initially allows setup of the accelerator parameters. Setting the gantry or the couch position causes movement commands to be issued; other accelerator parameters 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. A monitoring process periodically reads the accelerator status, i.e. the two dosimeter

readings. Normally, treatment continues until the prescribed dose has been delivered. However an incorrect dose, dose rate, or time limit will force treatment to be aborted. 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.

4.3 Test Annotations

Key specification events were annotated with PCL value constraints as follows:

```
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 readings *)
   MakeStatus(values(0,25,28),values(10,26,35)),        (* second readings *)
   MakeStatus(values(0,1,3),values(10,50,70)))          (* third readings *)
```

Most of the input values are simple ranges or typical values. The *MakeStatus* operation records a pair of dosimeter readings. As discussed in 2.2.2, the *accelerator* constraint defines three such pairs, used on each of three successive treatments. The dosimeter values are chosen to match the dose values, artificially introducing some variations in readings. The values cause treatment to stop on the final value of each triple. In the third treatment, it is supposed that the first dosimeter is incorrectly reporting low values; readings from the second dosimeter cause treatment to end.

The value constraints above are combined using the ordering constraints given as an example in section 2.2.3. The complete set of constraints is translated into LOTOS as outlined in section 2.3, adding about 180 lines to the basic specification.

4.4 Accelerator Test Generation

The TestGen tool was run to generate test cases based on the PCL annotations. The automaton initially generated has 41097 states and 62224 transitions. After minimisation with respect to observational equivalence, the automaton has 520 states and 546 transitions.

Exhaustive coverage of all paths through the automaton generates 256 test cases, of which the following is a sample. For brevity, outputs to the operator display have been omitted below. In this test, the operator initially sets the accelerator into electron mode and chooses to use an accessory. The operator then starts off three treatment cycles. At the beginning of each, the operator sets accelerator parameters and starts treatment. The control system then takes over, monitoring dosimeter readings until treatment is finished. At this point, a *Pass* verdict is recorded. As permitted by the optional ***alternate*** constraint, this particular test does not set the gantry, rotation, latitude, longitude and vertical parameters.

The reader should be able to match the test case below to the value constraints in section 4.3 and the ordering constraints in section 2.2.3.

```
                                                    (* start of test case *)
Console !Mode !ElectronMode                         (* operator sets electron mode *)
Console !Accessory !AccessoryIn                     (* operator chooses accessory *)

Console !Energy !6                                  (* operator sets energy 6 MeV *)
Console !Dose !5                                    (* operator sets dose 5 cGy *)
Console !Rate !1                                    (* operator sets dose rate 1 cGy/min *)
Console !CollimatorX1 !0                            (* operator sets collimator 1 x-coord *)
Console !CollimatorX2 !1                            (* operator sets collimator 2 x-coord *)
Console !CollimatorY1 !0                            (* operator sets collimator 1 y-coord *)
Console !CollimatorY2 !1                            (* operator sets collimator 2 y-coord *)
Console !Start                                      (* operator starts treatment *)
Accelerator !Set !MakeSetting(...)                  (* control system sets up accelerator *)
Accelerator !Start                                  (* control system starts treatment *)
Accelerator !Read !MakeStatus(2,2)                  (* control system reads dosimeters *)
Accelerator !Read !MakeStatus(1,1)                  (* and second pair of values *)
Accelerator !Read !MakeStatus(2,2)                  (* and third pair of values *)
Accelerator !Finish                                 (* accelerator reports treatment end *)
Console !Finished                                   (* operator told of treatment end *)

Console !Energy !13                                 (* operator sets energy 13 MeV *)
Console !Dose !52                                   (* operator sets dose 52 cGy *)
Console !Rate !25                                   (* operator sets dose rate 25 cGy/min *)
Console !CollimatorX1 !0                            (* operator sets collimator 1 x-coord *)
Console !CollimatorX2 !40                           (* operator sets collimator 2 x-coord *)
Console !CollimatorY1 !0                            (* operator sets collimator 1 y-coord *)
Console !CollimatorY2 !40                           (* operator sets collimator 2 y-coord *)
Console !Start                                      (* operator starts treatment *)
Accelerator !Set !MakeSetting(...)                  (* control system sets up accelerator *)
Accelerator !Start                                  (* control system starts treatment *)
Accelerator !Read !MakeStatus(0,10)                 (* control system reads dosimeters *)
Accelerator !Read !MakeStatus(25,26)                (* and second pair of values *)
Accelerator !Read !MakeStatus(28,35)                (* and third pair of values *)
Accelerator !Finish                                 (* accelerator reports treatment end *)
Console !Finished                                   (* operator told of treatment end *)
```

```
Console !Energy !20                    (* operator sets energy 20 MeV *)
Console !Dose !100                     (* operator sets dose 100 cGy *)
Console !Rate !50                      (* operator sets dose rate 50 cGy/min *)
Console !CollimatorX1 !39              (* operator sets collimator 1 x-coord *)
Console !CollimatorX2 !40              (* operator sets collimator 2 x-coord *)
Console !CollimatorY1 !39              (* operator sets collimator 1 y-coord *)
Console !CollimatorY2 !40              (* operator sets collimator 2 y-coord *)
Console !Start                         (* operator starts treatment *)
Accelerator !Set !MakeSetting(...)     (* control system sets up accelerator *)
Accelerator !Start                     (* control system starts treatment *)
Accelerator !Read !MakeStatus(0,10)    (* control system reads dosimeters *)
Accelerator !Read !MakeStatus(1,50)    (* and second pair of values *)
Accelerator !Read !MakeStatus(3,70)    (* and third pair of values *)
Accelerator !Finish                    (* accelerator reports treatment end *)
Console !Finished                      (* operator told of treatment end *)
                                       (* end of test case – Pass *)
```

At present, test cases like these have to be entered and executed manually on the accelerator. In future it is intended to convert test cases into prescription files. Prescriptions (i.e. pre-planned treatments) are normally devised by an oncologist using a separate treatment planning system. When the patient arrives for treatment, the prescription is automatically loaded into the accelerator. By handling test cases like prescriptions, it will be possible to execute them automatically. The accelerator logs all actions, so its response to a test case will be analysed offline by comparing the log and the test cases. The goal, of course, is to discover situations in which the accelerator does not behave as the specification requires. This is particularly critical after an upgrade of the accelerator software.

5 Conclusion

System specification with LOTOS has been briefly introduced. To have any practical hope of generating tests, the specification must be annotated with guidance as to useful test inputs. Although PCL has been designed to help with accelerator testing, it is generic and should be useful for testing in other domains. PCL annotations define key test inputs – explicit values (say, for an enumerated type) or boundary values (for a numeric range). Unconstrained events are also marked. PCL annotations are further used to constrain how inputs are ordered. The resulting constraint processes are automatically generated and placed in parallel with the main behaviour, allowing a manageable automaton to be generated.

The theory of input-output conformance is used to check whether an implementation agrees with its specification. A suspension automaton is generated from the LTS of the constrained specification. The suspension automaton is traversed to generate test cases that form a test suite. A transition tour may visit each edge at least once (for infinite behaviour) or may cover each path (for finite behaviour).

Radiotherapy accelerators have been briefly described. These are complex, software-controlled systems whose correct operation is vital for successful and

safe treatment of cancer. It is therefore very desirable to test their control systems systematically. A typical accelerator model has been outlined. PCL annotations have been given, along with an example of what the generated test cases look like. Test cases must currently be executed manually, though a strategy for automatic execution is being investigated.

The case study has demonstrated the following:

- that it is practicable to specify the key behaviour of radiotherapy accelerators using LOTOS
- that it is necessary to constrain the values in such specifications in order to make test generation practicable
- that PCL is adequate for constraining data-dominated specifications so that tests can be generated from them
- that the principles behind *ioconf* can be used to generate tests for radiotherapy accelerators.

Some important questions arise from the approach:

- Is the specification a faithful reflection of what an accelerator should do? In the main, the specification has been based on information from domain experts (the radiation physicists who oversee their operation). This is significant in that such experts see an accelerator as a black box. It would have been useful to gain insight into the detailed design of an accelerator, but attempts to involve an accelerator manufacturer have so far provided only limited information. The specification is believed to be a plausible model of an accelerator. However, more detailed experience with testing may show up flaws where the *specification* is incorrect, not the *implementation*.
- Are the test annotations appropriate? It may be that boundary value testing should be supplemented with other techniques that select critical values, e.g. determined by white-box knowledge of the implementation.
- Are the generated tests practicable? The current size of the test suite (256 cases) is manageable, though small variations in the test annotations can result in test suites from 16 to several thousand test cases. Small numbers of test cases (say, less than 20) can be manually executed. But for larger numbers of cases, an automated approach that simulates patient prescriptions will be essential.
- Are the generated tests useful? This is a much harder question to answer at this stage. Of necessity, test coverage is a tiny fraction of possible system behaviour due to the extensive use of data to control the accelerator. By concentrating on boundary value testing that is known to be useful in general software developing, it is hoped that the tests will be able to uncover problems.
- Can the tests discover known faults? Since the Therac-25, a record of accelerator problems has been built up. There have been incidents – fortunately rare and generally minor – since the original Therac-25 problems. Now that test suites can be generated and executed, it is intended to make a detailed study of what known faults can be found. Failure to discover such faults could arise from an error in the specification, an inappropriate choice for its level of abstraction, or a limitation of the strategy for generating selected test cases.

All these issues are being actively studied in ongoing work.

More theoretical techniques would also be an interesting future development. 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, and forbidding certain accelerator setups. Such properties could be used to check the integrity of the specification. It is conceivable that a hybrid solution could be devised, exploiting model checking results for both the specification and the implementation. Test generation based on symbolic values is also a promising line of enquiry.

Although this research 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 author is 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 author. Mr. Qian Bing collaborated on all of the work reported here. Dr. Ji He implemented most of the test generation tool. The author thanks Dr. Jan Tretmans (University of Nijmegen) for his insights into test generation.

References

1. G. Blair, L. Blair, H. Bowman, and A. Chetwynd. *Formal Specification of Distributed Multimedia Systems*. UCL Press, London, UK, 1998.
2. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1):25–59, Jan. 1988.
3. E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. K. Sabnani, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1988.
4. 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 (FORTE XIV)*, pages 184–200. Kluwer Academic Publishers, London, UK, Sept. 2001.
5. G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and verification of the PowerScale bus arbitration protocol: An industrial experiment with LOTOS. Technical Report 2958, INRIA, 78153 Le Chesnay Cedex, France, Aug. 1996.
6. R. G. Clark. The development of concurrent ADA systems from LOTOS specifications. In R. J. Mitchell and D. Simpson, editors, *ADA into the 90's*, pages 115–129. Woodhead Publishing Ltd, 1991.
7. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 2280 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2002.
8. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theory of Computer Science*, pages 83–133, 1984.
9. H. Eertink and D. Wolz. Symbolic execution of LOTOS specifications. In M. Diaz and R. Groz, editors, *Proc. Formal Description Techniques V*, pages 295–310. North-Holland, Amsterdam, Netherlands, Oct. 1992.

10. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1985.
11. M. Faci, L. M. S. Logrippo, and B. Stepien. Structural models for specifying telephone systems. *Computer Networks*, 29(4):501–528, Mar. 1997.
12. 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*, number 1102 in Lecture Notes in Computer Science, pages 437–440. Springer-Verlag, Berlin, Germany, Aug. 1996.
13. J. C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer-Verlag, Berlin, Germany, 1996.
14. M.-C. Gaudel and P. R. James. Testing algebraic data types and processes: A unifying theory. *Formal Aspects of Computing*, 10(5):436–451, 1999.
15. J. P. Gibson. A LOTOS-based approach to neural network specification. Technical Report CSM-112, Department of Computing Science and Mathematics, University of Stirling, UK, May 1993.
16. D. Greene and P. C. Williams. *Linear Accelerators for Radiation Therapy*. IOP Publishing Ltd., Bristol and Philadelphia, 1997.
17. 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.
18. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1985.
19. IEEE. *VHSIC Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.
20. IEEE. *IEEE Standard Hardware Design Language based on the Verilog Hardware Description Language*. IEEE 1364. Institution of Electrical and Electronic Engineers Press, New York, USA, 1995.
21. 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.
22. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework*. ISO/IEC 9646. International Organization for Standardization, Geneva, Switzerland, 1991.
23. ISO/IEC. *Information Technology – Framework: Formal Methods in Conformance Testing*. ISO/IEC 13245-1. International Organization for Standardization, Geneva, Switzerland, 1997.
24. 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.
25. ITU. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework*. ITU X.290. International Telecommunications Union, Geneva, Switzerland, 1996.
26. J. Jacky. Specifying a safety-critical control system in Z. In J. C. P. Woodcock and P. G. Larsen, editors, *Formal Methods Europe '93: (Industrial-Strength) Formal Methods*,

volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1993.
27. J. Jacky and M. Patrick. Modelling, checking and implementing a control program for a radiation therapy machine. In *Proc. AAS*. Dec. 1996.
28. J. Jacky and J. Unger. Formal development of A graphical user interface for a radiation therapy machine. In J. P. Bowen and M. G. Hinchey, editors, *Proc. 9th. International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, Sept. 1995.
29. 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*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, Dec. 1996.
30. C. Jard and T. Jéron. TGV: Theory, principles and algorithms. *Software Tools for Technology Transfer*, 2004. In this special issue.
31. 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, London, UK, Sept. 1999. Kluwer Academic Publishers.
32. Ji He and K. J. Turner. Specification and verification of synchronous hardware using LOTOS. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX)*, pages 295–312, London, UK, Oct. 1999. Kluwer Academic Publishers.
33. Ji He and K. J. Turner. Verifying and testing asynchronous circuits using LOTOS. In T. Bolognesi and D. Latella, editors, *Proc. Formal Methods for Distributed System Development (FORTE XIII/PSTV XX)*, pages 267–283, London, UK, Oct. 2000. Kluwer Academic Publishers.
34. E. J. Joyce. Accelerator linked to fifth radiation overdose. *American Medical News*, 1, Feb. 1987.
35. 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.
36. G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, Aug. 1992.
37. N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
38. N. G. Leveson, editor. *Safeware: System Safety and Computers*. Addison-Wesley, Reading, Massachusetts, USA, 1995.
39. A. McClenaghan. Experience of using LOTOS within the CIM-OSA project. In K. R. Parker and G. A. Rose, editors, *Formal Description Techniques IV*, pages 109–116, Amsterdam, Feb. 1992. North-Holland.
40. A. J. R. G. Milner. *Communication and Concurrency*. Addison-Wesley, Reading, Massachusetts, USA, 1989.
41. A. M. D. Moreira and R. G. Clark. Complex objects: Aggregates. Technical Report CSM-123, Department of Computing Science and Mathematics, University of Stirling, UK, May 1994.
42. D. Moundanos, A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47:2–14, 1998.
43. R. D. Nicola. External equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.
44. D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, Dec. 1990.

45. C. M. P. Reade. Process algebra in the specification of graphics standards. Technical Report CSTR-92-1, Department of Computer Science, Brunel University, Middlesex, UK, Sept. 1992.
46. J. M. T. Romijn, O. Sies, and J. R. Moonen. A two-level approach to automated conformance testing of VHDL designs. *Testing of Communicating Systems*, 10:432–447, 1997.
47. M. H. Thomas. The story of the Therac-25 in LOTOS. *High Integrity Systems Journal*, 1(1):3–15, Feb. 1994.
48. J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks*, 29:25–59, 1996.
49. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.
50. K. J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York, Jan. 1993.
51. K. J. Turner. Representing new voice services and their features. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 123–140. IOS Press, Amsterdam, Netherlands, June 2003.
52. K. J. Turner and Q. Bing). Protocol techniques for testing radiotherapy accelerators. In D. A. Peled and M. Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 81–96. Springer-Verlag, Berlin, Germany, Nov. 2002.
53. K. J. Turner, A. McClenaghan, and C. Chan. Specification and animation of reactive systems. In V. Atalay, U. Halici, K. İnan, N. Yalabik, and A. Yazici, editors, *Proc. International Symposium on Computer and Information Systems XI*, pages 355–364, Ankara, Turkey, Nov. 1996. Middle-East Technical University.
54. F. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. *IEEE Transactions on Very Large Scale Integration Systems*, 3:201–214, 1995.
55. C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. *Theoretical Computer Science*, 89:179–206, 1991.
56. I. Widya, F. Sadoun, and G.-J. van der Heijden. Specification of a distributed coordination function in LOTOS. In K. R. Parker and G. A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 133–148. North-Holland, Amsterdam, Netherlands, Nov. 1991.
57. K. Yasumoto, A. Kitajima, T. Higashino, and K. Taniguchi. Hardware synthesis from protocol specifications in LOTOS. In S. Budkowski, E. Najm, and A. Cavalli, editors, *Proc. Formal Description Techniques XI/Protocol Specification, Testing and Verification XVIII*. Chapman-Hall, London, UK, 1998.