Experience with Z
Developing a Control Program for a Radiation Therapy Machine

Jonathan Jacky [*], Jonathan Unger, Michael Patrick and Ruedi Risler

Radiation Oncology
Box 356043
University of Washington
Seattle, WA 98195-6043

Submitted to:
ZUM '97 Tenth International Conference of Z Users

September 19, 1996

Abstract

We are developing a control program for a unique radiation therapy machine. The program is safety-critical, executes several concurrent tasks, and must meet real-time deadlines. Development employs both formal and traditional methods: we produce an informal specification in prose (supplemented by tables, diagrams and a few formulas) and a formal description in Z. The Z description includes an abstract level that expresses overall safety requirements and a concrete level that serves as a detailed design, where Z paragraphs correspond to data structures, functions and procedures in the code. We validate the Z texts against the prose specification by inspection. We derive most of the code from the Z texts by intuition and verify it by inspection but a small amount of code is derived and verified more formally. We have produced about 250 pages of informal specification and design description, about 1200 lines of Z and about 6000 lines of code. Experiences developing a large Z specification and writing the program are reported, and some errors we discovered and corrected are described.

[*]email `jon@radonc.washington.edu`, telephone (206)-548-4117, fax (206)-548-6218

1 Introduction

This paper reports on the development of the control program for the therapy operator's console for a unique radiation therapy machine, including our use of formal methods with the Z notation.

(Authors' note to reviewers: At this writing (August 1996) the program is not yet complete and has not been placed in clinical use. We hope to have data from acceptance testing and clinical use in time for the final paper submission or the meeting itself.)

2 Purpose

This was not a pilot study or demonstration project. The purpose of the project was to develop the program that we will use to administer neutron therapy at our clinic. We anticipate using this program (or similar versions) for at least ten years, with more than a thousand patients.

The purpose of the therapy console program is to help ensure that patients are treated correctly, as directed by their prescriptions. The treatment console computer stores a database of prescriptions for many patients. Each patient's prescription usually includes several different beam configurations called *fields*. Each field is defined by about fifty machine settings (positions, dose etc.) that must be set properly to deliver the prescribed treatment. The console program enables the therapist to choose fields from the prescription database. The program sets some settings automatically, but others (external motions that present collision hazards) must be set manually by the therapist. The program checks all settings against the prescription and ensures that the radiation beam can only turn on when the correct settings for the chosen field have been achieved (subject to override by the therapist for some settings in some circumstances). The therapist can turn on the therapy beam (by a separate nonprogrammable mechanism) after the program indicates that the machine is ready.

The program provides a user interface (so the therapist can select prescriptions and view machine status) and controls devices. Low-level device control (such as turning the beam on and off and guiding machine motions) is performed by other nonprogrammable mechanisms, programmable logic controllers (PLC's), and simple embedded computers. The therapy console program provides some of these low-level controllers with endpoints (such as positions and doses), and it enables (or disables) motions and activation of the beam.

This program is just one component of a large control system that includes other pro-

grams, several computers and many non-programmable elements (for example, interlocks implemented in "hard-wired" relay logic). The delegation of functions among the software and hardware components was a prerequisite to the work discussed here and is described elsewhere [9, 10].

2.1 Safety

The program is safety-critical because it could contribute to delivering a treatment that differs from the prescribed one, irradiating the wrong volume within the patient or delivering the wrong dose. We rely on nonprogrammable mechanisms (physical machine design, hardware interlocks, etc.) to establish generic safety conditions that are the same for every treatment and do not depend on the prescription. Moreover, therapists can visually check most settings before they turn on the therapy beam. However, we have to place some reliance on our program to establish and check some safety conditions that are different for each prescription, including field shape and dose.

In our machine the physical design (not the control program) limits the radiation dose to medically reasonable values. In this respect our machine differs from the radiation therapy machines that delivered severe overdoses in several computer-related accidents [16]. However we do rely on the control program to set and monitor the positions of several field shaping filters in the treatment head which are practically invisible to the therapists and are downstream from the machine's internal dose monitors. Errors in these functions could contribute to delivering doses up to three times larger or smaller than prescribed, which could result in treatment failure or serious complications.

2.2 Concurrency

The program must cope with concurrency because it handles multiple devices: several low-level controllers and the therapist's console terminal. The program may have to wait for input from one or more of these devices at the same time that it must be proceeding with other activities. Moreover, some devices (including the console) can signal unsolicited events that demand a prompt response.

2.3 Real time

The system is designed so the most stringent timing requirements are met by hardware or by embedded computers which are only indirectly controlled from the therapy console.

The therapy control program itself has no hard real-time requirements, although it should respond to most events (by updating the display or sending commands to controllers) reasonably promptly (within a few tenths of a second). There are also some timeouts and expirations with deadlines ranging from tenths of seconds to hours.

2.4 Platform

The therapy console program runs on a commercially available microcomputer and was created using an embedded software development product [2]. There is no separate operating system nor any other software on the therapy console microcomputer other than the control program. The programming language is a proprietary Pascal dialect that provides support for multiprocessing and device control [3].

The user interface (display and keyboard) is handled by the X window system [20], programmed using `Xlib` [18] only. The therapy console terminal is an X terminal that is not running a window manager nor any other applications.

2.5 Clinical environment

The program we developed is a replacement for an older therapy console program that was developed by the therapy machine manufacturer and has been in use since the machine was installed in 1984 [19]. The therapy machine is not a conventional linear accelerator that provides electron and X-ray beams; it uses a cyclotron to produce a neutron beam. It has an excellent record of safety and reliability [15] and produces better clinical results than conventional machines for some cancers. The computer controls are an essential part of the system and have had dramatic clinical impact because complex shaped fields are necessary to avoid unacceptable complications in neutron therapy [1]. This unique machine is maintained and upgraded by our department, not the manufacturer.

Although our program replaces an earlier program that had to meet similar requirements, it is entirely new. It presents a different user interface to the therapist, and some of the equipment that it controls has also been changed (the hardware changes were made while the software development project was underway). The new program runs on a different computer and uses a different programming language and system software. There is no reuse of code, design or specifications from the earlier program. In fact we have no detailed specification or design for that program; we believe that reconstructing them would have required almost as much work as designing the new program.

The original program has worked well, but when it was first delivered the manufacturer had

to make many revisions at our site to correct errors and improve usability before it could be used clinically. It took several months. This time we will not have the opportunity to remove the machine from clinical service for an extended debugging period.

3 Method

3.1 Personnel and project management

The project was performed by the authors, who are full-time technical staff members in the Radiation Oncology department, a clinical department at the University of Washington Medical Center. All have advanced degrees and had years of experience with radiation therapy, accelerator engineering or software development before joining the project. None had experience with formal methods before this project.

Risler is chief engineer of the facility and and is responsible for the overall system. The software was developed by a team of two people (Jacky throughout, working with Unger, then Patrick). We have full technical responsibility for the project; there is no separate quality assurance group or certifying body.

3.2 Programming methodology

The formal methods we used in this project are not a new or radical approach for us. They are an incremental improvement that fits well into a method of working that we have used for many years on other projects (we prefer to call it a "method of working" because "development method" or "methodology" often connotes the use of a particular notation and design technique; we choose notations and techniques according to the needs of each project).

Our method is to consider carefully what the program's capabilities should be and how it should work, record our decisions in documents, and then write the program itself from the contents of the documents. The primary quality assurance method is to review the documents, and review the code against the documents. An essential feature of the method is our willingness to rewrite or discard documents and code that seem unclear or overly complicated, whether or not we discover any errors [14]. We also test thoroughly from a written test plan [13], but testing is not our primary quality assurance technique; we have found that it is a waste of time to test code that has not passed review.

Formal notations (mathematical and logical formulas) are not an essential element of our method. Our choice of notation is pragmatic: we use whatever mix of prose, formulas, tables and diagrams best support review and code development. The therapy console is the first project where we have used a formal notation that can be checked by machine (other than the code itself).

The particular set of documents we produce is usually different for each project. Documents are usually not written and completed in sequence; most documents are begun early in the project and are revised at frequent intervals as the project proceeds.

Besides documents, we also create prototype programs to demonstrate to the users and to become familiar with the platform and programming environment. Prototypes may not be documented or reviewed and are eventually discarded.

The following subsections describe the products (documents and code) we produced for this project.

3.2.1 Informal specification

The informal specification [9, 10, 7] describes the system in prose, diagrams, tables and a few formulas. It is explicit and very detailed. The authors and reviewers of these documents include the physicist who defined the physical and clinical requirements for the original machine, engineers who installed and maintain the machine, and clinical users of the machine. These documents are the only products of the development that were written (in part) and reviewed by users who are not software developers. They comprise the real specification in this project because they record what the machine's users require from the software developers.

3.2.2 Formal description

We created a formal description [8] in Z notation [22]. The Z texts express a detailed design. We used Z to discover the design, not just to document an already existing design. There are few descriptions of other therapy control systems (see [16, 23]) and these do not provide enough detail to serve as examples. We had to create our own. We used no other design notation (except prose).

It is a bit misleading to call the Z texts a specification; the prose documents are that. The formal texts are actually a model (a simplified, abstracted representation) of portions of the control program itself. Z variables correspond to program variables and data structures,

Z operation schemas correspond to procedures, etc. The model is very detailed: each X window system event (including every keystroke) and transmission or receipt of every message to or from a controller is modelled by a Z operation schema. Some samples from the Z texts appear in section 4.5 below.

The informal specifications are sufficiently explicit and detailed to determine the required behaviors, but they are intended to support comprehension and review by the users, not to guide programming. Therefore, we (software developers) created the formal description exclusively for our own use, to guide coding and review. We use it as a kind of documentation; we were attracted by the conciseness and expressivity of the formal notations; we did not anticipate doing proofs or formally deriving code.

The formal description was written to facilitate review. In our report [8] each Z paragraph is accompanied by prose commentary that includes cross references to pertinent section, page numbers (and often paragraph and line numbers) in the informal specification (chapter 8 in [10]). The report also contains a glossary that identifies Z variables with items described in the informal specification.

We examined our Z texts with a type checker [21]. We also tried a well-formedness checker [17] (which checks for undefined expressions).

Jacky made the decision to produce a formal description when the informal specification was already partly written. He learned Z for this purpose and wrote most of the formal description. Unger and Patrick joined the project later; both learned Z on the job and also contributed to the formal description.

3.2.3 Implementation guide

We wrote a document that describes how we implemented the program [12]; it contains additional information that is needed to review the code. It explains our use of the platform's system software, nonstandard extensions in the programming language, the X window system, and describes some of our programming techniques. It names and describes each program source file and explains the dependencies between files.

3.2.4 Code

All software developers wrote code from Z specifications. We derived most of the code directly from the Z texts by intuition and verified it by inspection, without any intermediate formal refinement steps. Z variables were implemented by program variables and data

structures, Z operation schemas were implemented by procedures, etc. The Z texts were written to make these translations obvious. We wrote down *post hoc* correctness arguments in a few cases where the intuitive derivation was not obvious to all.

In one instance we did a formal derivation of about one page of code from the specification (see [11], also section 4.1.3 below).

3.2.5 Tests

We perform tests by following instructions in written test plans called *scripts* that describe exactly to how the execute the program and what the results should be (example scripts from another project appear in [13]). We script three kinds of tests: *Functional tests* attempt to cover the specification in some systematic way. *Stress tests* check the program's response to erroneous or unusual conditions involving hardware and files. *Acceptance tests* are simulated treatment sessions (with no patient) that rehearse both typical and unusual clinical situations.

Before the program was complete, each of us compiled and executed code, discovered errors, and made corrections ad lib. This activity was not scripted or recorded.

4 Results

4.1 Products

We wrote about 250 pages of pertinent informal specification and design description, about 1200 lines of Z and about 6000 lines of code[1]. Table 1 reports the exact sizes of the products (documents and code) described in subsections 3.2.1 through 3.2.5. The following subsections describe each product.

4.1.1 Informal specification

The informal specification comprises three separate documents. Part I [9] is an overview and summary, Part II [10] specifies the user interface, and Part III [7] describes hardware interfaces and external file formats. They are 106, 235 and 131 pages long, respectively (11

[1] Our estimates for the completed project. Table 1 reports the sizes at this writing.

PRODUCT	SIZE	
Informal specification: overview, entire facility [9]	106 pages	
Informal specification: user interface, entire facility [10]	235 pages	
Informal specification: user interface, therapy only (in [10])	45 pages	
Informal specification: hardware and files, therapy only [7]	131 pages	
Formal description (Z texts) [8]	77 pages	(1137 lines)
Implementation Guide [12]	42 pages	
Program code	4786 lines	(41 files)
Test scripts	35 pages	

Table 1: Development products (documents and code)

point type, single spaced). Parts I and II describe the entire system including the cyclotron; the therapy console user interface is described in a single 45-page chapter in Part II (this can also serve as the users' reference manual). All of Part III is devoted to the therapy console.

4.1.2 Formal description

The formal description [8] expresses the therapy console behaviors described in the informal specification [10, 7]. Some samples from the Z texts appear in section 4.5 below.

There are 176 pages of relevant informal description, but most of the formal description focusses on only 45 pages (chapter 8 only in [10]). The formal description comprises 1137 lines of Z (207 paragraphs, including 131 schema definitions), presented in a 77 page report [8] (most of this report is prose)[2].

The formal texts were validated by inspection: declarative sentences in the informal specification correspond to predicates (mostly about set membership and values of function applications) which should be (mostly obvious) consequences of predicates in the formal specification. In a handful of cases we wrote down a few derivation steps.

All software developers participated in the reviews and all discovered and corrected errors where the Z formulas did not express the intended behaviors. Some errors in the Z were not discovered until the coding stage; writing code necessarily involves intensive review.

[2]The line count is the number of nonblank lines output by running the Fuzz tool `-v` option [21] on the report. This output is similar to the LaTeX source for the Z formulas.

CATEGORY	Z	PROOF	CODE
Process and event handling	40	40	433
Pervasive constants and types	73	—	200
File handling and persistent data	52	—	672
Operations and volatile data	764	20	747
Graphics utilities	—	—	589
Graphics displays	—	—	1233
Low-level device control	208	—	912
Total	1137	60	4786

Table 2: Lines of formal description, proof, and code

The formal description was rewritten several times, partly to correct errors in content but mostly to improve its organization and make it easier to validate and use as a guide for coding. We believe this indicates the difficulty inherent in creating a good design, rather than difficulties with formal methods or the Z notation.

The type checker [21] detected many trivial errors; we corrected them. The well-formedness checker [17] found a few undefined expressions (function applications) which were deliberate elisions, not oversights. We did not do any machine-checked proofs to confirm that the formal texts express the intended behaviors.

4.1.3 Code

The code can be classified into categories defined in the implementation guide [12]. Table 2 shows the lines of Z description, code[3], and any derivation or proof in each category. The table reveals that we modelled different parts of the program at very different levels of detail, according to our judgment about the novelty and difficulty of each portion. Some portions of the program have no formal description at all, while the formal description of some portions is as large as the code itself.

We didn't anticipate doing any formal derivation or verification, but we did a little anyway. As is customary in most Z literature, we did not (at first) write a formal specification for the main program that invokes the various operation schemas, but of course we had to implement one. This code is invoked each time an X window event (such as a keypress) occurs. We saw opportunities to achieve efficiencies in testing preconditions, but implementing our ideas turned out to be more difficult than we expected. When our intuitively written code

[3]Noncomment, nonblank lines of code

grew complicated and we could not convince ourselves that it was correct, we decided not to resort to testing and debugging but to attempt a formal derivation (in axiomatic style) instead. This quickly exposed an outright error in our intuitively written code and also revealed that it was needlessly complicated, so we abandoned it. We are using the formally developed code (it is about a page long). This development is reported in [11].

An early version of the program was based on the formal description and implemented most of the required behaviors. It was not intended to be a prototype and worked reasonably well, but we found it difficult to review and saw many opportunities for improvement, so we discarded it.

4.1.4 Test scripts

We scripted some functional tests from the Z descriptions. The behavior of the program is described by a collection of Z operation schemas. These can be shown in a kind of state transition table where each row in the table shows an operation, the first column shows the part of the precondition involving only state variables, the second column shows the precondition involving input variables, and the third column shows the operation itself, which determines the next state (see examples in [11] and [8]). When the third column of one row implies the first column of another, the two operations can occur in sequence. This makes it possible to script sequences of operations and check that various coverage measures have been achieved (for example that each operation has been executed, or that all possible pairs of consecutive operations have been executed).

4.2 Development Effort

The informal specification [9, 10] was developed over about five years. This effort included the entire facility including the cyclotron, not just the therapy console. The therapy console products, including the formal description [8], other documents [7, 12] and code was developed over about four years. In this latter effort, all products except the informal description [7] was developed by a team of two people (Jacky throughout, working with Unger, then Patrick) who often had obligations on other projects as well. The products were not begun and finished in sequence; all products were revised throughout the entire four years.

We (very roughly) estimate that we have devoted about six person-years of effort to developing the therapy console program. This includes learning the platform and its programming environment, learning Z, and producing prototypes, documents and code that we later discarded.

4.3 Comparable projects

The original control program that ours replaces comprises about 12,000 lines of code (FORTRAN).

While they were working on the therapy console program, Jacky and Unger also participated in another project that expended similar effort but produced a much larger program: in about seven person-years it produced about 300 pages of documentation and 40,000 lines of code (Lisp) [14]. We feel the difference indicates the greater difficulty of producing a safety-critical embedded control system compared to a scientific application that runs on standard workstations, rather than the difficulty of formal methods.

Before beginning work on the therapy console description [8], Jacky wrote some smaller Z descriptions of portions of the cyclotron controls [4, 5, 6] (106, 166 and 178 lines of Z, respectively); these have not yet been implemented. The therapy console description was much more difficult and required far more effort than the increase in size might suggest.

4.4 Experience testing and using the program

The main program code discussed in section 4.1.3 received the most intensive formal development [11]. When we began functional testing this code failed almost immediately. The error resulted from an omission in the formal specification, which was based on a poorly chosen (overly restricted) example. The error did not arise in any formal development step so it could not have been detected by checking these steps more intensively, but it might have been detected by more thorough validation of the specification. It was easy to correct the omission and repeat each step in the formal development. After this single correction we have not found any more errors in this code.

A few errors in the Z texts (where they did not express the intended behaviors) were not discovered until ad-hoc testing.

(At this writing (August 1996) we have not yet begun stress testing, acceptance testing or clinical use.)

4.5 Design

We used the Z notation to discover a design; this section discusses some features of the design we chose.

The central idea of the therapy control program is this safety requirement: the beam can only turn on when the actual state or *setup* of the machine is physically safe, and matches a *prescription* that the operator has selected and approved. We must only deliver setups that are physically consistent and reasonable or *safe*. The control program helps ensure that we can only treat a patient when the *measured* machine setup *matches* a *prescribed* setup.

$$[SETTING, VALUE, FIELD]$$

$$SETUP == SETTING \longrightarrow VALUE$$

$$safe\_ : \mathbb{P}\, SETUP$$
$$match\_ : SETUP \leftrightarrow SETUP$$
$$prescription : FIELD \nrightarrow SETUP$$

SafeTreatment

$$measured, prescribed : SETUP$$

$$safe(measured)$$
$$match(measured, prescribed)$$
$$prescribed \in \operatorname{ran} prescription$$

The whole design arises from elaborating this simple model. The entire purpose of the control program is to establish and confirm the *SafeTreatment* condition. The *prescribed* setup must be selected; the *measured* setup must be achieved; the *safe* and *match* conditions must be tested.

Although this model seems simple, the informal description of the actual machine is complicated. There are lots of operations and each one involves many different subsystems. Without careful design, the implementation also threatens to be complicated, difficult to review, and error prone.

We surmised that the apparent complexity of the informal specification arises from the interaction of several subsystems which, by themselves, are simpler. We eventually realized that the obvious partition into the subsystems that the users see is not the best one. Instead of a "vertical" partition with subsystems for the leaf collimator, the dosimetry system etc. we chose a "horizontal" partition with with (for example) a subsystem for settings (including the settings for the leaf collimator, the dosimetry system and all the rest), another for all interlocks etc. The vertical alternative requires proliferation of similar items and operations in each subsystem, while the horizontal alternative turns out to be simpler and more uniform overall.

The *Field* schema includes the state variables that represent settings for the currently selected *field*. Sensors report *measured* setting values. *Prescribed* setting values are read

from the prescription database. There are some additional details: for example, some settings that do not match their prescribed values can be *overridden* by the operator. It is necessary to store the value of each setting when it is overridden.

Field

$$field : FIELD$$
$$measured, prescribed : SETUP$$
$$overridden : SETTING \pfun VALUE$$

$$field \in \mathrm{dom}\, prescription$$
$$prescribed = prescription\, field$$

The *Intlk* schema declares state variables that model interlocks and status flags. The *status* function indicates the readiness of each setting. *Interlocks* prevent undesired or potentially hazardous situations. The function *intlk* indicates the status of each interlock; operations are inhibited when interlocks are *set* and are enabled when interlocks are *clear*. The master therapy interlock *therapy_intlk* (not included in *intlk*) is a special software-controlled interlock that must be clear to allow the beam to turn on. Clearing the master therapy interlock is the central safety-critical act of the program.

$[INTERLOCK]$

$INTLK ::= clear \mid set$

$READY ::= ready \mid not\_ready \mid override$

Intlk

$$therapy\_intlk : INTLK$$
$$intlk : INTERLOCK \fun INTLK$$
$$status : SETTING \fun READY$$

These schemas suggest an implementation where the Z given sets *SETTING* and *INTERLOCK* are implemented by enumerated types whose values are the actual setting and interlock names, and the Z functions *measured*, *prescribed*, *overridden*, *intlk* and *status* are implemented by arrays indexed by these enumerations. Settings and interlocks that belong to the same hardware subsystem are assigned to subranges (consecutive values) in their respective enumerations, so operations that apply to a single hardware subsystem are implemented by loops that iterate over these subranges.

Having partitioned our system and described the parts separately, we must compose the parts together again. In our partitioned design the *measured* and *prescribed* setups appear in the *Field* subsystem, while the software-controlled interlocks appear in the *Intlk* subsystem. The *SafeTreatment* state schema sketched earlier includes both subsystems.

SafeTreatment

Field
Intlk

. . . predicate omitted . . .

The program sets and clears the interlocks, including the master therapy interlock, by periodically executing the *ScanIntlk* operation.

ScanIntlk

Ξ*Field*
Δ*Intlk*

$therapy\_intlk' =$ **if** *SafeTreatment* **then** *clear* **else** *set*

Here the *SafeTreatment* state schema is implemented by a function that tests the schema predicate. Ξ*Field* expresses that the implementation of this operation only requires read-only access to variables in *Field*.

5 Project status and further work

(Authors' note to reviewers: At this writing (August 1996) the program is not yet complete and has not been placed in clinical use. However most of the user interface and some device control functions are working. Many operations can be performed and we believe the code now in place will not require major revisions. We hope to have data from acceptance testing and clinical use in time for the final paper submission or the meeting itself.)

We hope to reuse some products of this effort in future projects (the cyclotron operator's console program, etc.).

Our use of formal methods in this project was largely a paper-and-pencil effort emphasizing description rather than analysis. However our formal texts may be able to support more intensive, machine-supported analyses. Can they detect significant errors or suggest useful improvements that we have missed?

6 Conclusions

We found the formal notation useful for discovering a design, and then documenting the detailed design such that it could be validated against the prose specification, and could

also serve as a guide for writing and inspecting the code. We can make some observations based on our experiences so far:

- Formal methods can help create novel designs and develop original code. They are not just for documenting existing designs and analyzing code that has already been written.
- It is very difficult to produce a useful formal description that faithfully expresses the informal requirements for a complex system and can also serve as a basis for developing code. There are problems of scale and organization and not much guidance from the small examples in the literature.
- A detailed and explicit informal specification that has been reviewed by the systems' designers and users (not just software developers) is an indispensable prerequisite to any use of formal methods. Only this can serve as the standard for validation. It is a major portion of the whole project effort, not just a preliminary.
- A useful formal description is not just a paraphrase of the informal specification into mathematical notation. Creating the formal description requires design judgment in addition to understanding the requirements and the formal notation.
- All documents and code require much revision for clarity and organization, not just content and correctness.
- Software developers who have the education and experience needed to work on this kind of project can learn to read, review, and implement Z and even write small amounts of it fairly quickly. Writing a useful formal description of a complex system is much more difficult and requires much experience on progressively harder problems.
- Simply having a good formal description does not guarantee that a good implementation will come easily. Diligent ongoing review is required to ensure that the implementation is simple and clear enough to review against the formal description. This is a prerequisite to checking that the implementation is correct.
- Some errors will escape detection in reviews so testing is still an essential technique for discovering errors.

Acknowledgments

The authors thank Mark Saaltink for running the well-formedness checks and thank Ira Kalet for assistance with the project.

References

[1] Mary Austin-Seymour, Richard Caplan, Kenneth Russell, George Laramore, Jon Jacky, Peter Wootton, Sharon Hummel, Karen Lindsley, and Thomas Griffin. Impact of a multileaf collimator on treatment morbidity in localized carcinoma of the prostate. *International Journal of Radiation Oncology Biology Physics*, 30(5):1065–1071, Dec 1994.

[2] Digital Equipment Corporation, Maynard, Massachusetts. *Introduction to VAXELN*, October 1991.

[3] Digital Equipment Corporation, Maynard, Massachusetts. *VAXELN: Pascal Programming Guide*, December 1991.

[4] Jonathan Jacky. Formal specifications for a clinical cyclotron control system. In Mark Moriconi, editor, *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pages 45–54, Napa, California, USA, May 9–11 1990. (Also in *ACM Software Engineering Notes*, 15(4), Sept. 1990).

[5] Jonathan Jacky. Formal specification and development of control system input/output. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop, London 1992*, pages 95–108. Proceedings of the Seventh Annual Z User Meeting, Springer-Verlag, Workshops in Computing Series, 1993.

[6] Jonathan Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, 1995.

[7] Jonathan Jacky, Michael Patrick, and Ruedi Risler. Clinical neutron therapy system, control system specification, Part III: Therapy console internals. Technical Report 95-08-03, Radiation Oncology Department, University of Washington, Seattle, WA, August 1995.

[8] Jonathan Jacky, Michael Patrick, and Jonathan Unger. Formal specification of control software for a radiation therapy machine. Technical Report 95-12-01, Radiation Oncology Department, University of Washington, Seattle, WA, December 1995.

[9] Jonathan Jacky, Ruedi Risler, Ira Kalet, and Peter Wootton. Clinical neutron therapy system, control system specification, Part I: System overview and hardware organization. Technical Report 90-12-01, Radiation Oncology Department, University of Washington, Seattle, WA, December 1990.

[10] Jonathan Jacky, Ruedi Risler, Ira Kalet, Peter Wootton, and Stan Brossard. Clinical neutron therapy system, control system specification, Part II: User operations. Technical Report 92-05-01, Radiation Oncology Department, University of Washington, Seattle, WA, May 1992.

[11] Jonathan Jacky and Jonathan Unger. From Z to code: A graphical user interface for a radiation therapy machine. In J. P. Bowen and M. G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation*, pages 315–333. Ninth International Conference of Z Users, Springer-Verlag, 1995. Lecture Notes in Computer Science 967.

[12] Jonathan Jacky, Jonathan Unger, and Michael Patrick. CNTS implementation. Technical Report 96-04-01, Department of Radiation Oncology, University of Washington, Box 356043, Seattle, Washington 98195-6043, USA, April 1996.

[13] Jonathan Jacky and Cheryl P. White. Testing a 3-D radiation therapy planning program. *International Journal of Radiation Oncology, Biology and Physics*, 18:253–261, January 1990.

[14] Ira J. Kalet, Jonathan P. Jacky, Mary M. Austin-Seymour, Sharon M. Hummel, Kevin J. Sullivan, and Jonathan M. Unger. Prism: A new approach to radiotherapy planning software. *International Journal of Radiation Oncology, Biology and Physics*, 36(2):451–461, 1996.

[15] Ira J. Kalet, Jonathan P. Jacky, Ruedi Risler, Solveig Rohlin, and Peter Wootton. Integration of radiotherapy planning systems and radiotherapy treatment equipment: 11 years experience. *International Journal of Radiation Oncology, Biology and Physics*, 38(1):213–221, 1997.

[16] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[17] Irwin Meisels and Mark Saaltink. The Z/EVES reference manual. Technical Report TR-95-5493-03, ORA Canada, 267 Richmond Road, Suite 100, Ottawa, Ontario K1Z 6X3 Canada, December 1995.

[18] Adrian Nye. *Xlib Programming Manual.* O'Reilly and Associates, Inc., Sebastopol, CA, 1988.

[19] Ruedi Risler, Jüri Eenmaa, Jonathan P. Jacky, Ira J. Kalet, Peter Wootton, and S. Lindbaeck. Installation of the cyclotron based clinical neutron therapy system in Seattle. In *Proceedings of the Tenth International Conference on Cyclotrons and their Applications*, pages 428–430, East Lansing, Michigan, May 1984. IEEE.

[20] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[21] J. M. Spivey. *The fuzz Manual.* J. M. Spivey Computing Science Consultancy, Oxford, second edition, July 1992.

[22] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, New York, second edition, 1992.

[23] Martin S. Weinhous, James A. Purdy, and Conrad O. Granda. Testing of a medical linear accelerator's computer-control system. *Medical Physics*, 17(1):95–102, Jan/Feb 1990.