Modelling, Checking and Implementing a Control Program for a Radiation Therapy Machine

Jonathan Jacky and Michael Patrick

Radiation Oncology, Box 356043
University of Washington
Seattle, WA 98195-6043
jon@radonc.washington.edu

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. We have produced about 250 pages of informal specification and design description, about 1200 lines of Z and about 6000 lines of code. 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 express requirements in the abstract level that we would like to check in the concrete level; checking these properties automatically could reveal subtle errors in the detailed design. We selected a small portion of our design (13 transitions, 227 lines of Z, corresponding to 331 lines of code) and performed a pencil-and-paper (non-automated) analyses that revealed several errors. We translated this portion of our Z model to the SMV input language. The translation requires considerable simplification, where Z predicates and numerical, structured variable values are encoded as small integers. We repeated our analyses using the SMV model checker. The automated analyses detected the known errors (we did not detect any additional errors in our corrected version).

Introduction

We are developing a new control program for a unique radiation therapy machine. This is not a pilot study or demonstration project; its purpose is to develop the program that we will use to administer therapy at our clinic.

The program is safety-critical; errors could contribute to irradiating the wrong volume within the patient or delivering the wrong dose. We have produced about 250 pages of informal specification and design description, about 1200 lines of Z [9] and about 6000 lines of code [8].

Our principle assurance method thus far has been careful (but informal and subjective) review of documents and code. This has been reasonably effective but a few design errors present in the Z have escaped detection until testing [8] so we would welcome automated analysis. To determine whether this could be feasible for us, we consider the formal description of our system.

Formal description

Our formal description comprises 1137 lines of Z (207 paragraphs, including 131 schema definitions), presented in a 77 page report [6] (simplified excerpts appear in [4]). The Z description has two levels: an abstract level that expresses overall safety requirements and a concrete level that serves as the detailed design. The requirements level expresses properties that we wish to check or prove; the design level is the system to be checked.

The abstract level expresses the central idea of the therapy control program, which 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.

Most of our Z texts are devoted to the concrete level, which expresses a design. The design 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. The translation to code is almost obvious. Some excerpts from this level appear below.

What we wish to check

Critical requirements can be expressed in Z at our abstract level. Using the *SafeTreatment* schema defined above and another schema *BeamOn* (not defined here) we can express the central safety requirement that the beam can only be on when the machine is in a safe condition:

$$BeamOn \Rightarrow SafeTreatment$$

Checking this property would be a significant achievement. It would provide additional evidence (besides inspection and testing) for the soundess of our detailed design. It would not be a superficial exercise; it should reveal even low-level errors because our design is so detailed.

Checking this property appears to be beyond the capabilities of any existing checker. Nitpick [3] can check single operations on Z data structures (but not numbers) and temporal logic model checkers such as the Symbolic Model Verifier (SMV) [2] can check sequences of operations on simple data. However we need to check sequences of operations on data structures (including numbers).

The requirements we wish to check involve rather complicated predicates on the numerical values of a large amount of highly structured data. We cannot express these requirements directly in the input language of contemporary checkers such as Nitpick or SMV because they are too inexpressive and place many restrictions on data (for example variables cannot have numerical values, or can have only small integer values).

It is possible to translate our requirements from Z to a checker input language in order to perform some particular analysis, but the translation involves considerable simplification, where the predicates and numerical, highly structured data values we really care about are encoded as small integers or enumeration values.

For example, much of our system can be modelled by a state transition system with not too many states, but the states are not given to us *a priori*; each state in the transition system is actually a (possibly very large) set of states where the values of some collection of variables satisfies some predicate. In order to create the checker input and interpret the results from the checker, it is necessary to first identify the states in a more expressive notation such as Z.

The relation that exists between the checker input and the more expressive Z model can be expressed by the familiar Z refinement relations [9], except here development proceeds from concrete to abstract rather than vice-versa.

An experiment in model checking

To gain experience in model checking we selected a small portion of our detailed design (13 transitions, 227 lines of Z, corresponding to 331 lines of code) where pencil-and-paper (non-automated) analyses had initially revealed several errors. We translated this portion of our Z model to the SMV input language and repeated our analyses using the SMV model checker. The automated analyses detected the known errors (we did not detect any additional errors in our corrected version). The following sections describe the experiment in more detail.

System description

We considered one subsystem of our control program: the process the communicates with a separate embedded controller called the *treatment motion controller* (TMC). Our program exchanges messages with the TMC and several other controllers to perform operations specific to our application, such as setting up the the therapy machinery to conform to a patient's prescription [5, 7].

The TMC controller process handles *events* (in our implementation these events derive from hardware and software interrupts); the process may be *running* or *waiting* for an event. The process and its controller exchange *messages*. Messages can be classified into types called *commands* (from the process) and *responses* (from the controller). Certain events called *command events* are invoked by the user to signal the controller process to issue a whole sequence of commands that accomplish some useful goal, such as automatically setting up machine components to prepare for a treatment. The process responds by sending that sequence of commands to the controller; for each of these commands, a particular sequence of responses is ex-

pected. In addition to command events, there is a *receive* event that occurs when a response message appears. It is also possible to arrange for a *timeout* event to signal that a deadline has expired.

We had to confront several design issues: the first is *pending commands*. The user interface process that generates command events runs independently of each controller process, and it can take appreciable time to execute the sequence of commands that can be elicited by a single command event. When a command sequence is still in progress, several more command events might be signalled. We decided that a command sequence in progress must run to completion; it cannot be pre-empted. However new command events that occur while a command sequence is in progress are not discarded, they are stored and then handled in turn. A command event which has been signalled but not handled is said to be *pending*. Each command event is assigned a priority, and pending command events are handled in priority order (not arrival order). Moreover, only one instance of each kind of command event can be pending; if an impatient user repeatedly signals the same pending command event, it will be handled only once.

The next issue is *polling*. In addition to handling command events signalled by the user, the process must frequently *poll* its controller: command it to report the most recently measured values of pertinent machine settings. We decided that a process which is not executing a command sequence and has no pending commands should request a timeout; if the timeout expires, the controller executes the polling command sequence. As a result, a process which is not busy will poll periodically. Alternatively, if a command event occurs the timeout is cancelled and polling is deferred while the command event is handled instead. This is acceptable because it is not necessary for polling to be strictly periodic.

A third issue concerns *error handling*. A controller which does not obey its protocol is exhibiting erroneous behavior and might present a hazard. It is important for the process to detect controller errors quickly, so appropriate protective action can be taken [1]. For each command that the process issues, the protocol determines which response is expected; if a different (or unintelligible) response appears, the process indicates a controller error. Moreover, every response is associated with a deadline; if the deadline expires and no response has arrived, a timeout is signalled and the process indicates a controller error. Finally, the process indicates an error if it receives an unsolicited message from the controller. When a controller error is detected, any command sequence in progress is abandoned, and pending command events are cancelled, except for *restart* events that are provided to resume normal operations after replacing or powering up a controller.

Our prototype implementation of the TMC controller comprised 331 (non-blank, non-comment) lines of code in a Pascal dialect that provides support for multitasking and device control (the entire control program contains about 6000 lines of code). We performed a few successful tests that confirmed we understood the basic operations of the controller.

Z model

We modelled the functions provided by our initial implementation in 227 lines of Z. The Z texts include thirteen operation schemas that each define one state transition in the controller process.

The state of each controller process is the set of unhandled events, the real time clock, a timer to store the deadline when a timeout is pending, the (inferred) status of the controller, the processing state, the sequence of pending commands that have not yet been sent to the controller, and the sequence of responses expected for the most recently sent command that have not yet been received. This is expressed by the *Controller* state schema:

$$STATUS ::= ok \mid error$$

$$PROCESSING ::= wait \mid run$$

Controller

$$events : \mathbb{P}\, EVENT$$
$$clock, timer : TIME$$
$$status : STATUS$$
$$processing : PROCESSING$$
$$pending : \mathrm{seq}\, COMMAND$$
$$expected : \mathrm{seq}\, RESPONSE$$

The environment outside the *Controller* state provides different events, which we model as different values of the input variable $e?$. When an event occurs, it joins the set of unhandled events. Modelling unhandled events as a set ensures there can be no more than one unhandled event of each kind. No other state variables change when an event is signalled. This state transition is expressed by the *Signal* operation schema:

Signal

$$\Delta Controller$$
$$e? : EVENT$$

$$events' = events \cup \{e?\}$$
$$\ldots$$

The two basic controller operations are *Wait* and *HandleEvent*. The controller process executes

[1] There are also hardware protection mechanisms that work independently of the control program.

Wait to wait for an event. The processing state switches from *run* to *wait*.

Wait

$\Delta Controller$

$processing = run$
$processing' = wait$
$\ldots$

When an awaited event is signalled, the controller process executes *HandleEvent*. *HandleEvent* changes *processing* from *wait* to back to *run*; the controller process alternates executing *Wait* and *HandleEvent*. Specializations of *HandleEvent* are called *handlers*; most of the work of the controller process is accomplished by handlers.

Each handler waits for a particular event; if this event appears and the handler's other preconditions are also satisfied, the handler will remove this event from the set and execute. Events accumulate in a set, not a queue; if there is more than one unhandled event, any enabled handler might run. Determinism can be introduced by restricting handlers' preconditions.

HandleEvent

$\Delta Controller$
$e? : EVENT$

$e? \in events$
$processing = wait$
$processing' = run$
$events' = events \setminus \{e?\}$
$\ldots$

In the initial state, there are no pending commands or expected responses. The controller process waits for a command event.

WaitCmd

$Controller$

$pending = \langle\rangle$
$expected = \langle\rangle$

After the user causes a command event to be signalled, *HandleCmd* loads the appropriate sequence of new commands. If the user does not signal a command event, the timeout itself acts as a command event: it invokes the polling command sequence. If the controller is not *ok* this handler is disabled, except for certain *restart* commands which are provided to recover from the error state.

HandleCmd

$HandleEvent$
$c : COMMAND$

$WaitCmd$
$e? \in cmd$
$status = ok \lor e? \in restart$
(**let** $cs == commands\ e?$ •
$\quad c = head\ cs \land pending' = tail\ cs$)
$status' = status$

The *Send* operation creates a message from the command and (if appropriate) the data.

Send

$\Delta Controller$
$c : COMMAND$
$data? : DATA$
$message! : MESSAGE$

$message! = \ldots$ format msg from c and $data?$
$expected' = responses\ c$

These two operations are combined to send a new command

$$NewCommand \mathrel{\widehat=} HandleCmd \land Send$$

After executing *NewCommand* the process resets the timer for the appropriate deadline and waits for the controller to respond.

$$WaitReceive \mathrel{\widehat=} [Controller \mid expected \neq \langle\rangle]$$

When a handler detects a possible controller error, it sets the controller state to *error* and abandons any command sequence in progress.

ControllerError

$HandleEvent$

$state' = error$
$pending' = \langle\rangle$
$expected' = \langle\rangle$

Paper-and-pencil analysis

It is necessary to gain understanding of a design before beginning automated analyses. Simple paper-and-pencil analyses can be quite effective at this stage. They revealed the properties we checked during the automated analyses.

Each of our thirteen Z operations defines one state transition. To show how the operations work together we collect the pertinent formulas in one place. Table 1 is a state transition table, similar to the mode transition tables of the SCR notation [1].

There is one entry in the table for each Z operation schema. The four columns in the table show the operation name, the conjuncts of the precondition that only involve the state variables,

Z operation name	State precondition	Input precondition	Progress postcondition
$Wait$	$processing = run$	$true$	$processing' = wait$
$\boxed{StartTimer}$	$WaitCmd$	$true$	$timer' = clock + period$
$RestartTimer$	$WaitReceive$	$true$	$timer' = clock + deadline$
$HandleEvent$	$processing = wait$	$e? \in events$	$processing' = run$ $events' = events \setminus \{e?\}$
$NewCommand$ $(HandleCmd \land Send)$	$WaitCmd \land p$	$e? \in cmd$	$WaitReceive'$ $expected' = responses\ c$ $pending' = tail\ cs$
$CommandError$	$WaitCmd \land \neg\ p$	$e? \in cmd$	$status' = error$
$ReceiveUnsolicited$	$WaitCmd$	$e? = receive$	$status' = error$
$ReceiveAck$	$WaitReceive$ $tail\ expected \neq \langle\rangle$	$e? = receive \land q$	$status' = ok$ $expected' = tail\ expected$
$NextCommand$ $(ReceivePending \land Send)$	$WaitReceive$ $tail\ expected = \langle\rangle$ $pending \neq \langle\rangle$	$e? = receive \land q$	$status' = ok$ $expected' = responses\ c$ $pending' = tail\ pending$
$ReceiveComplete$	$WaitReceive$ $tail\ expected = \langle\rangle$ $pending = \langle\rangle$	$e? = receive \land q$	$WaitCmd' \land status' = ok$ $expected' = tail\ expected$
$ReceiveError$	$WaitReceive$	$e? = receive \land \neg\ q$	$WaitCmd' \land status' = error$
$ReceiveTimeout$	$WaitReceive$	$e? = timeout$	$WaitCmd' \land status' = error$
$Signal$	$true$	$true$	$events' = events \cup \{e?\}$
$SignalEvent$	$true$	$e? \neq timeout$	
$SignalTimeout$	$clock > timer$	$true$	$timer' = stopped$
$Tick$	$true$	$true$	$clock' > clock$

$WaitReceive \Leftrightarrow expected \neq \langle\rangle$
$WaitCmd \Leftrightarrow pending = \langle\rangle \land expected = \langle\rangle$
$WaitCmd \lor WaitReceive$ is invariant; $pending \neq \langle\rangle \land expected = \langle\rangle$ is forbidden.

$c = head\ pending \land cs = tail(commands\ e?)$
$p \Leftrightarrow status = ok \lor e? \in restart$; $\langle cmd, \{receive\}\rangle$ partition $EVENT$
$q \Leftrightarrow message? \in \mathrm{dom}\ response \land response\ message? = head\ expected$

Table 1: Controller operations, preconditions, and progress postconditions

the conjuncts of the precondition that only involve input variables, and conjuncts from the postcondition that indicate a change of state (postconditions of the form $x' = x$ indicating that x does not change are not shown). In addition to the thirteen top-level operations, there are entries for three building-block operations. Each group of top-level operations is headed by the building-block operation whose predicates are conjoined with theirs. For example, *Wait* appears above *StartTimer* so the full state precondition of *StartTimer* is $processing = run \wedge WaitCmd$.

Many properties of our protocol can be confirmed by inspecting Table 1:

Invariance: Every operation has either *WaitCmd* or *WaitReceive* in its precondition. If the controller process entered a state where neither predicate were true, no operations would be enabled; the process would deadlock. Therefore $WaitCmd \vee WaitReceive$ must be an invariant. The negation of the invariant is the forbidden state where an unsent command is pending but no reply is expected.

Completeness: The protocol definition is complete if the response to all events is defined in all states permitted by the invariant. The table shows this is true: the disjunction of all the preconditions forms a tautology. For example in the second column, there are two large sections for $processing = run$ and $processing = wait$, together these cover all values of *processing*. Within each section there are subsections for *WaitCmd* and *WaitReceive*, together these cover the invariant $WaitCmd \vee WaitReceive$, etc.

Determinism: The protocol is deterministic if only one operation is enabled for each input in each state. The table shows that all the preconditions are disjoint. For example in the second column we find the disjoint pairs $processing = run$ and $processing = wait$, *WaitCmd* and *WaitReceive*, etc.

Progress (liveness): To make progress, it is necessary to make state transitions from one table entry to another. For example unhandled command events can accumulate in the *WaitReceive* state. However all the operations that begin in *WaitReceive* either end in *WaitCmd* or make progress toward *WaitCmd* by establishing $expected' = tail\ expected$ or $pending' = tail\ pending$. Therefore the system will eventually reach *WaitCmd* and pending command events will be handled.

Errors

In this study we consider three errors we discovered in early versions of our Z model, while we were preparing Table 1. There were two *missing operations*: we forgot *RestartTimer*, and we only provided a single *Receive* operation instead of *ReceiveAck*, *ReceiveComplete* and *NextCommand*. There was also a *displaced precondition*: $pending = \langle\rangle$, intended to prevent a new command event from pre-empting a command sequence that is already in progress, appeared in *HandleEvent* not *WaitCmd*.

SMV model

We expressed our model in the input language of the SMV checker, obtaining 124 (nonblank, noncomment) lines of SMV code. SMV is less expressive than Z so we had to create a simpler model that still represents the interesting behaviors. For example we modelled message contents (data) in Z but in SMV we only modelled message type; validity of message contents was modelled by a separate nondeterministic boolean input. In Z we modelled pending commands and expected responses by sequences; in SMV we only modelled the lengths of the sequences. In SMV we modelled time very coarsely; the clock is reset on each *Wait* operation and operations time out if the expected response does not appear on the next tick.

Once we determined how to simplify our model, translating from Z to SMV was not difficult. We declared SMV variables corresponding to each Z state variable. Using Table 1 as a guide, we defined SMV symbols corresponding to useful Z state schemas: *WaitCmd* becomes `WaitCmd := pending = 0 & expected = 0`. Then we defined an SMV symbol for the precondition of each operation, not forgetting to conjoin the preconditions from the heading: for *StartTimer* we obtain `preStartTimer := preWait & WaitCmd`. Finally, we wrote SMV assignment statements to accomplish the state changes that appear in the rightmost column of the table. For each state variable there is a case statement. For each operation where the variable changes value, there is a case branch that assigns the new value, guarded by that operation's precondition. For example *StartTimer* is implemented by `next(timer) := case ... preStartTimer: period; ...`.

A model expressed in the SMV input language is called an SMV program. Figure 1 shows excerpts from our SMV program that deal with the two state variables *pending* and *expected*.

SMV analysis

An SMV program is analyzed by submitting it to the checker, along with claims expressed as formulas in Computation Tree Logic (CTL). The checker either verifies each claim or produces a counterexample: a sequence of states generated by the program that violates the claim.

First we checked our translation into SMV by the method of Atlee and Gannon [1]. For each of our thirteen operations, we submitted two claims: `EF(`*precondition*`)` checks that states

```
MODULE main

VAR
   ...
   pending: {0,1,2};                -- length of seq. of pending commands
   expected: {0,1,2};               -- length of seq. of expected responses

DEFINE
   ...
   WaitReceive := expected > 0;      -- expected \neq <>
   WaitCmd := pending = 0 & expected = 0; -- pending = <> /\ expected = <>
   ...
   preReceive := preHandleEvent & WaitReceive & ReceiveEvent;
   preReceiveAck := preReceive & expected > 1 & q;
   preNextCommand := preReceive & expected = 1 & pending > 0 & q;
   preReceiveComplete := preReceive & expected = 1 & pending = 0 & q;
   preReceiveError := preReceive & !q;
   preReceiveTimeout := preHandleEvent & WaitReceive & TimeoutEvent;

ASSIGN
   ...
   init(pending) := 0;              -- pending = <>
   init(expected) := 0;             -- expected = <>

   ...
   next(pending) := case
        -- Model command sequences of differing lengths in this simple way:
        -- timeout event elicits sequence of one command, setup elicits two.
        -- The new value of pending is the length of the *tail* of the sequence
        preNewCommand & TimeoutEvent: 0; -- NewCommand: pending' = tail cs
        preNewCommand & SetupEvent: 1; -- NewCommand: pending' = tail cs
        preNextCommand: pending - 1; -- NextCommand: pending' = tail pending
        preReceiveError: 0;         -- ReceiveError: WaitCmd'
        preReceiveTimeout: 0;       -- ReceiveTimeout: WaitCmd'
        1: pending;                 -- otherwise, no change
   esac;

   next(expected) := case
        preNewCommand: 1;           -- NewCommand: expected' = responses c
        preNextCommand: 2;          -- NextCommand: expected' = responses c
        preReceiveAck: expected - 1; -- ReceiveAck: expected' = tail expected
        preReceiveComplete: expected - 1;
                                 -- ReceiveComplete: expected' = tail expected
        preReceiveError: 0;         -- ReceiveError: WaitCmd'
        preReceiveTimeout: 0;       -- ReceiveTimeout: WaitCmd'
        1: expected;                -- otherwise, no change
   esac;
```

Figure 1: Excerpts from SMV program based on Table 1.

that satisfy the operation's precondition are reachable, and `AG(`*precondition* `-> AF(`*postcondition*`))` checks that executing the operation results in the intended postcondition. It is necessary to check both claims; if the `EF` "liveness" claim is false, the `AG` claim will be vacuously true. At first the checker found counterexamples to several of these claims, which we traced to several trivial translation errors. After we corrected the errors, all these claims were verified.

Next we considered how to detect errors. Usually we do not know which errors are present, so how can we form claims that will detect them? We reasoned that it is best to begin with simple claims about system invariants or progress properties because these claims probe every operation, not just a few.

Our paper-and-pencil analysis revealed that *WaitCmd* $\vee$ *WaitReceive* should be true in every state. In CTL this invariant is expressed `AG (WaitCmd | WaitReceive)`. If any of our thirteen operations ever fails to maintain the invariant, this claim is false. The simplest progress property is that a waiting process will eventually run. In CTL this is `AG (!run -> AF run)`. If it is ever possible for the process to deadlock, this claim is false. Neither claim is explicit in the model; both were discovered during the pencil-and-paper analysis. The checker verified both claims for our SMV program.

We produced three erroneous versions of our SMV program, each seeded with one of the errors we discovered in our early Z models. The checker found counterexamples to the progress claim for all three versions (all three deadlocked). The checker verified the invariance claim for two of the erroneous versions, but found a counterexample in the version where the *NextCommmand* operation was missing.

The checker consumed negligible resources checking these claims, using at most 0.16 seconds of system time, 10201 BDD nodes, and 983040 bytes (on an HP 715/50 workstation with 32MB of memory). The longest counterexample was a sequence of 18 states.

Conclusion

Model checking promises to be more conclusive than trial-and-error methods such as testing or experimenting with prototypes. In order to fulfil this potential it is necessary to have meaningful claims to check. Claims should express the central requirements of the application, should be easy to validate against an intuitive understanding of the requirements, and errors anywhere in the model should be likely to generate counterexamples. Safety requirements, global invariants and progress properties are examples of such claims.

References

[1] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[2] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. De Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, pages 124 – 175, Noordwijkerhout, The Netherlands, 1993. REX School/Symposium, Springer-Verlag. Lecture Notes in Computer Science, vol. 803.

[3] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[4] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997.

[5] 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.

[6] 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.

[7] 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.

[8] Jonathan Jacky, Jonathan Unger, Michael Patrick, David Reid, and Ruedi Risler. Experience with Z developing a control program for a radiation therapy machine. In M. G. Hinchey, editor, *ZUM '97*. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science (in press).

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