Lessons from the Formal Development of a Radiation Therapy Machine Control Program

Jonathan Jacky

1 Introduction

Formal methods work best where traditional programming methods don't work very well: problems that are too difficult to solve by intuition or too novel to solve by modifying some existing program or design. We are developing the control program for the therapy operator's console for a unique radiation therapy machine. This is not a pilot study or demonstration project; we will use this program to administer neutron therapy at our clinic. The program is safety-critical; 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 [1]. Errors in medical software can have serious consequences [11, 24, 23], and our experience shows that development methods typically used in this field are only partially effective at eliminating errors [21]. We decided to try a more rigorous method.

Moreover, our machine is a unique resource that justifies an unusual development effort. It is not a conventional therapy linear accelerator that provides electron and X-ray beams; it uses a cyclotron to produce a neutron beam, which produces better clinical results than conventional machines for some cancers [6]. 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 [3]. This unique machine is maintained and upgraded by our department, not the manufacturer. The program we are developing is a replacement for a program that was developed by the therapy machine manufacturer and has been in use since the machine was installed in 1984 [22]. There is no reuse of code, design or specifications from the earlier program.

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

2 What the program does

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. The delegation of functions among the software and hardware components was a prerequisite to the work discussed here and is described elsewhere [16, 17].

3 What the project produced

We wrote about 250 pages of informal specification and design description, about 1200 lines of Z and about 6000 lines of code[2]. Table 1 reports the size of each product.

The informal specification [16, 17, 14] describes the system in prose, diagrams, tables and a few formulas. Part I [16] is an overview of the entire facility, part II [17] specifies the user interface for every console (including accelerator operators' consoles) and part III [14] describes hardware interfaces and external file formats for the therapy console program only.

The formal description [15] is expressed in the Z notation [27]; its contents are discussed in sections 4 and 5 below. It comprises 1137 lines of Z (207 paragraphs, including 131 schema

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

PRODUCT	SIZE	
Informal specification: overview, entire facility [16]	106 pages	
Informal specification: user interface, entire facility [17]	235 pages	
Informal specification: user interface, therapy only (in [17])	45 pages	
Informal specification: hardware and files, therapy only [14]	131 pages	
Formal description (Z texts) [15]	77 pages	(1137 lines)
Implementation Guide [19]	42 pages	
Program code	4786 lines	(41 files)
Test scripts	35 pages	

Table 1: Development products (documents and code)

definitions), presented in a 77 page report [15] (most of this report is prose)[3]. Concurrency is handled in Z by the interleaving model recommended by Evans [5]: the state is shared by all processes, but operations can be invoked by different processes. The real time clock (needed for timeouts) is modelled in Z as an ordinary state variable.

Most of our Z texts express a detailed design: a collection of state variables, operations on those variables, and a partition of the variables and operations into modules. We used Z to discover the design, not just to document an already existing design. There are few descriptions of other therapy control systems and these do not provide enough detail to serve as examples [28]. We had to create our own. We used no other design notation (except prose). We also wrote an implementation guide in prose [19] to explain our use of the platform's system software and other design information not expressed in the Z texts.

The program is coded in ANSI C; concurrency and device control is provided through EPICS [4], a library originally developed for controlling research accelerators, which is built on a commercially available real-time embedded software development product. The user interface (display and keyboard) is handled by the X window system, programmed using `Xlib` only.

We derived most of the code directly from the Z texts by intuition and verified it by inspection, without any intermediate formal refinement steps. The Z design is so detailed that most of the translation is obvious. Z basic types and free types become enumerations. Z state variables become program variables and data structures. Z operation schemas become functions and procedures. A Z state schema and the operation schemas on that state become a module (in C, an `.h` file and a `.c` file that define a collection of constants, types and variables and the functions that use them). Z schema inclusion becomes module

[3]The line count is the number of nonblank lines output by running the Fuzz tool `-v` option [26] 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

inclusion (C `#include` directives). Some examples appear in [12].

The code can be classified into categories defined in the implementation guide. Table 2 shows the lines of Z description, code[4], 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.

To prepare for this project, Jacky wrote some smaller Z descriptions of portions of the accelerator controls [8, 9, 10] (106, 166 and 178 lines of Z, respectively); these have not yet been implemented.

The program is not yet complete and we have not begun acceptance testing. Tables 1 and 2 report the work accomplished as of this writing (May 1997). Additional information about project organization, staff, assurance methods, and level of effort appears in [20].

4 Writing the formal description

We found it quite difficult to produce a formal description that faithfully expresses the informal requirements and also serves as a useful basis for developing code. There are problems of scale and organization and not much guidance from the small examples in the literature. After many revisions we obtained a satisfactory solution by applying the following strategies.

[4]Noncomment, nonblank lines of code

4.1 Write strong invariants

The advantage of writing specifications in Z (or some similar notation) instead of code (or another operational notation) is that Z can describe states directly. Usually, fundamental requirements and the designer's intent are most naturally expressed as states (or sets of states): *assumptions* and *trigger conditions* can both be formalized as *preconditions*, *goals* become *postconditions*, and *safety requirements* become *invariants*. Code and other operational notations can only express these concepts indirectly (as the result of executing sequences of transitions).

It is possible to specify a system as nothing more than a collection of state transitions (Z operation schemas). Such specifications closely resemble code (in Z the "after" state would be given explicitly in equations that resemble assignment statements: $x' = \ldots$ etc.)[5]. In such transition-oriented specifications, transitions often seem arbitrary and unmotivated (because there is nothing else in the formal specification that they can be derived from or checked against). In fact no transition should be arbitrary; every transition should *make progress towards a goal* while *preserving the invariants*, but merely describing each transition (as assignments, or implicitly with pre- and postconditions) cannot express this. A specification in this style does not take advantage of Z; it conveys essentially the same information as the code itself. It may be more convenient or readable than the code, but it cannot support any deep analyses because it does not directly express the designers' intent.

Analysis of such transition-oriented specifications is limited to checking generic properties that are independent of any particular application, such as completeness, consistency, and determinism [7, 1][6]. These analyses can reveal some errors, but any generic analysis is necessarily weak because it cannot check the application-specific properties that are of greatest interest. A system that is complete, consistent and deterministic may nevertheless fail to satisfy its requirements.

In contrast to this transition-oriented style, Z encourages specifiers to write explicit invariants that can express the designers' intent more directly. Invariants convey additional information which is not present in the code and cannot be unambiguously derived from it (or "reverse engineered"). This additional information supports deep analyses that can reveal whether or not the specified transitions actually achieve the designers' intent. Each proposed transition can be checked to confirm that it makes progress toward the goal while preserving the invariants. In fact, it is sometimes possible to derive the transitions formally because transitions often contain two kinds of assignments: assignments to independent variables that make progress, and assignments to dependent variables that maintain in-

[5]In Z it is also possible to describe states implicitly. For example, we can specify an integer square root operation by $x' * x' \leq a < (x' + 1) * (x' + 1)$.

[6]Anderson et al. [1] did check some application-specific properties, but these were conjectures posed by the analysis team. We believe that system designers should provide the properties to be checked.

variants. In typical informal developments, the latter (dependent) assignments are often handled in a hit-or-miss way because programmers must rely on intuition to discover them. When invariants are explicit, dependent assignments can be inferred (for example see section 4.1.2 below).

In summary, our strategy for using Z is to write strong state invariants, to check every transition against the invariants, and to derive dependent assignments in transitions from invariants. In the following subsections we illustrate this strategy at two levels of abstraction: an abstract level that expresses overall system safety requirements and a concrete level that expresses the detailed design. In general, the requirements level expresses properties that we might wish to check or prove; the design level is the system to be checked.

4.1.1 System safety invariants

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.

$$[SETTING, VALUE, FIELD]$$

$$SETUP == SETTING \longrightarrow VALUE$$

$$\begin{array}{|l}
safe_ : \mathbb{P}\, SETUP \\
match_ : SETUP \leftrightarrow SETUP \\
prescription : FIELD \nrightarrow SETUP
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SafeTreatment}} \\
\hline
measured, prescribed : SETUP \\
\hline
safe(measured) \\
match(measured, prescribed) \\
prescribed \in \operatorname{ran} prescription \\
\hline
\end{array}$$

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. The predicates $safe(measured)$, $match(measured, prescribed)$, and $prescribed \in \operatorname{ran} prescription$ each act as goals for certain operations or collections of operations.

Essential safety requirements can be expressed at this level. For example, the condition of the machine when the radiation beam is on is expressed by a state schema, *BeamOn* (its predicate acts as another goal). The safety requirement that the beam can only be on when the machine is in a safe condition can be expressed

$$BeamOn \Rightarrow SafeTreatment$$

Checking or proving 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. We have not yet attempted this, but it is a long term goal of our work.

Our formal description includes more than 1000 lines of Z, but each line is motivated by this abstract model and can be traced back to it. We believe that designers should be able to express the central idea of their system formally on a single page, as we have done here.

4.1.2 Detailed design invariants

At the detailed design level Z state variables correspond to data structures in the code and Z operations correspond to procedures in the code. At this level we try again to write strong state invariants. For example our *Session* state models program data structures that represent the prescription database and aspects of the treatment session that are derived from it. At this level we must model the prescription database in greater detail. There are actually two collections of stored setups: one for patients' prescriptions, another for medical physics and radiobiology experiments which are performed with no patient present. The machine can be operated in two modes, one for patients and the other (with fewer safety interlocks) for the experimental studies. The control program maintains a list of names for which the operator can select setups. This list must be kept consistent with the operating mode: patients in therapy mode, studies in experiment mode. This consistency requirement is expressed by an invariant in the *Session* state.

$$[NAME]$$

$$\mid\ patients, studies : \mathbb{P}\, NAME$$

$$MODE ::= therapy \mid experiment$$

$$
\begin{array}{|l}
\hline
\textit{Session} \\
mode : MODE \\
names : \mathbb{P}\, NAME \\
\ldots \\
\hline
names = \mathbf{if}\ mode = therapy\ \mathbf{then}\ patients\ \mathbf{else}\ studies \\
\ldots \\
\hline
\end{array}
$$

Here the declarations in the Z (above the line) are intended to be translated to declarations of data structures in the code. However the invariant (below the line) does not correspond directly to any code, rather it expresses a property that the program should possess.

The *ExptMode* operation toggles the session from therapy mode to experiment mode or back again. It could be written:

$$
\begin{array}{|l}
\hline
\textit{ExptMode} \\
\Delta Session \\
\hline
mode' = \mathbf{if}\ mode = therapy\ \mathbf{then}\ experiment\ \mathbf{else}\ therapy \\
\ldots \\
\hline
\end{array}
$$

The predicate on $mode'$ is intended to be translated to an assignment statement. However something is missing. The invariant requires that the list of names must change whenever the mode changes. We should write the operation this way:

$$
\begin{array}{|l}
\hline
\textit{ExptMode} \\
\Delta Session \\
\hline
(mode', names') = \mathbf{if}\ mode = therapy\ \mathbf{then}\ (experiment, studies) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (therapy, patients) \\
\ldots \\
\hline
\end{array}
$$

Here *mode* is the independent variable; this operation makes progress by changing *mode*. In this operation *names* is the dependent variable; it must be changed to maintain the invariant. The predicate on $names'$ instructs the programmer that the code must read the appropriate file contents into control program data structures when the mode changes. This is the kind of thing which is often overlooked when programming from intuition.

In Z both versions of *ExptMode* mean exactly the same thing; writing out the predicate about $names'$ is redundant because it is implied by the invariant. We always try to use the second form; we feel strongly that it is important to write out the predicate for each state variable explicitly, even when it could be derived from the invariant. We write our code based on these operation schemas, and we feel it is unreasonable to derive consequences of invariants at coding time — that work should be part of specification, not implementation.

We also believe that it is very valuable to express the same requirement in two places, as an invariant in the state schema and as an assignment in the operation schema. We believe that using redundancy in this way is an important advantage of the Z notation and a key component of good Z style. It encourages specifiers to think about the requirements in a systematic way, and it gives reviewers something to look for.

Perhaps tools could help: given the value of the independent variable, the tool could calculate the value of the dependent variable. For example, given only $S \mathrel{\widehat=} [\ldots \mid x = a \Rightarrow y = b]$ and $Op \mathrel{\widehat=} [\Delta S \mid x' = a]$ with no explicit constraint on y' in Op, the tool should provide a (highly automated) reminder that $y' = b$ in Op.

4.2 Partition the design

Much of the apparent complexity of the therapy machine arises from the interaction of several subsystems which, by themselves, are simpler. We partition the system into subsystems and describe one state schema and several (or many) operation schemas on each. For each operation on the system as a whole, we define a separate operation on each affected subsystem. The complex behaviors of the whole system emerge when we compose these simpler operations together. In Z we perform the composition by schema composition or inclusion.

The advantages of this approach arise because many operations involve only a few of the subsystems, and many complex operations can emerge when simpler operations appear together in different combinations. As a result, the formal description of the partitioned design is shorter and clearer than would be possible with a monolithic design, and is easier to review in sections.

Identifying the best partition requires much exploration; we eventually discovered that the obvious partition corresponding to the hardware subsystems that the users see is not the best one. We settled on a partition that included these subsystems: *Session* models those aspects of the treatment session that are related to the prescription database, *Field* models the many settings (from all hardware subsystems) that characterize a single field, *Intlk* models software interlocks and other flags that indicate readiness, *Console* models the user interface, *TMC* models the process that communicates with the treatment motion controller (one of the embedded computers), etc. This partition can itself be expressed in Z.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad TherapyControl \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Session \\
Field \\
Intlk \\
Console \\
TMC \\
\ldots \\
\hline
\ldots \\
\hline
\end{array}
$$

Each subsystem has its own invariants, and as we combine subsystems we add more. For example the *Console* state schema represents which screen design or *display* is visible on the operator's console. *Console* also includes *nlist*, a list of items which may appear on the display.

$$[DISPLAY]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Console \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
nlist : \mathbb{P}\, NAME \\
display : DISPLAY \\
\ldots \\
\hline
\ldots \\
\hline
\end{array}
$$

One of the displays is the **Select Patient** display. This display shows the list of patients or experimental studies, so the operator may select one. To express this, we combine *Console* and *Session* and add an invariant.

$$
\begin{array}{|l}
select\_patient : DISPLAY
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ConsoleSession \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Console \\
Session \\
\hline
display = select\_patient \Rightarrow nlist = names \\
\ldots \\
\hline
\end{array}
$$

It is clear from this that if the operating mode is changed when the patient list display is visible, the display contents must change to show the list of experimental studies instead of patients (or vice versa). This is another example of something that might be missed if left to intuition.

4.3 What is an operation?

Z is just mathematics; designers must choose an interpretation for the mathematics that is meaningful for their project. It is necessary to decide what a Z operation schema represents, which entails answering these questions: What occurences in the real system will be modelled by operation schemas in the Z description? How do you know when your formal description is complete; that is, you have written all of the operations? The answers to these questions are not provided by the Z notation itself; designers must answer them for each project.

We decided that our choice of state variables determines what the operations must be: *any* occurence that changes the value of *any* state variable should be modelled by an operation schema. The formal description is not complete until all such operation schemas have been written. As a result of this decision, our description 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. Many activities that the users see as single operations, which are described as single operations in the prose specification, must be modelled in Z by several, or many, operation schemas. For example the activity described in the prose as the single **Select Field** (from the prescription database) operation is represented in Z by one operation to display the list of available fields, a second operation to move the cursor from one field on the list to the adjacent field, and a third operation to select the field under the cursor and update pertinent state variables with new values copied or derived from the prescription database. Moreover, this third operation is defined by composing several different operation schemas, one for each of the subsystems discussed in section 4.2.

One might ask, is all this low-level detail really necessary? We believe that it is. In order to understand errors that have caused fatal accidents, it is sometimes necessary to descend to this level [11, 24, 23]. The effort might seem burdensome, but it makes it possible to do analyses that can detect low-level errors.

4.4 When is an operation invoked?

Having chosen the operations, it is still necessary to decide how and when the implementation of each operation schema should be invoked. Z has no built-in concept of a program or any explicit control structure, and Z users often do not write any formal definition of the "main program" or "top level" that invokes the appropriate operation schemas when they are needed. Nevertheless, we have to implement the top level anyway.

We decided that our program is defined implicitly by the "top-level" operation schemas.

We can tell which operation schemas are at the top level because they are not included in any others. The control structure is determined implicitly by the input variables and preconditions of the top-level operations. Each top-level operation should be invoked whenever its input appears and all of its preconditions are satisfied. We had to code the machinery to make that happen. We defined the machinery formally, and the formal definition shows the way to the implementation. If we call the top level schemas $Op1, Op2, \ldots, OpN$ then the formal definition of the main program is just the disjunction of these operations.

$$Main \mathrel{\widehat{=}} Op1 \lor Op2 \lor \ldots \lor OpN \lor Exception$$

It is a good idea to include an *Exception* operation to handle any cases that might have been overlooked. The precondition for *Exception* is implicit in the program logic: control reaches *Exception* when no other operations are invoked.

An obvious implementation strategy would code the main program as a large case analysis (`switch` or `if` ... `else if` ... in C.) [12]. We chose instead to build a table-driven dispatcher. To add new operations we do not have to revise the program control structure, we merely add entries to a table and write functions that implement the operations. This code is derived and proved correct in [18].

Our program implements a particular interpretation of the meaning of the precondition of an operation. In our view, preconditions are *guards* or *triggers*: an operation is invoked when its input appears and all of its preconditions are satisfied. For example, consider again the *ExptMode* operation that switches between therapy and experiment mode (section 4.1.2, above). Experiment mode is only permitted to certain privileged operators called *physicists*. We model this by adding another state variable and an invariant to the *Session* state.

$$[OPERATOR]$$

$$physicists : \mathbb{P}\ OPERATOR$$

Session

$$\begin{array}{l}
\ldots \\
operator : OPERATOR \\
\ldots \\
\hline
\ldots \\
mode = experiment \Rightarrow operator \in physicists \\
\ldots
\end{array}$$

A physicist may switch operating modes by pressing a function key (modelled by a value of the *INPUT* type).

$[INPUT]$

$$expt\_mode : INPUT$$

$$
\begin{array}{l}
\textit{ExptModeOp} \\
\hline
\Delta ConsoleSession \\
ExptMode \\
input? : INPUT \\
\hline
input? = expt\_mode \\
operator \in physicists \\
operator' = operator \\
\ldots
\end{array}
$$

The precondition of *ExptModeOp* is $input? = expt\_mode \land operator \in physicists$. This is a *trigger*: our program invokes the *ExptModeOp* operation whenever the **Experiment Mode** key is pressed and the operator is a physicist. If the operator is not a physicist, pressing this key does not cause any change of state (in our implementation it does elicit an output: a message indicating that an unprivileged user may not select experiment mode).

In Z, the precondition $operator \in physicists$ is redundant because it can be inferred from the invariant $mode = experiment \Rightarrow operator \in physicists$[7]. Nevertheless, we always try to write out the precondition explicitly. We take advantage of the opportunity to express the same requirement in two places, as an invariant in a state schema and as a guard in an operation schema. We would consider the omission of $operator \in physicists$ from *ExptModeOp* to be an error. A tool that can calculate implicit preconditions such as [25] could be used to detect such errors.

Our interpretation of preconditions is a design decision, it is not determined by the Z notation itself. Another interpretation holds that preconditions in Z are correctness conditions, not triggers (this interpretation may be more common than ours). In this view it is possible to invoke an operation even when the precondition is not satisfied, but in that case the operation may not behave as specified. According to this view, it is valid for the precondition of the implementation of an operation to be weaker than the precondition of its specification. However this not always valid when preconditions are interpreted as triggers.

[7]This inference is valid in the case where $mode = therapy \land mode' = experiment$ but not vice-versa.

5 Checking the formal description

The initial draft of any nontrivial formal description will be full of errors. It is necessary to have some systematic method for detecting and correcting them. In our project we used (subjective) reviews with very little automated support. We discovered a great many errors and eventually produced a much better revised description. We continue to find minor errors almost every time we reread the revised formal description thoroughly; despite this, is quite usable as our primary source for coding and test planning.

We did not do any formal proofs to confirm that the Z texts express the intended behaviors. We did use a type checker [26] and we corrected the numerous but trivial errors that it found. We also tried the domain checking capabilities of a theorem prover [25]. The prover found a few expressions where functions were applied to arguments that could not be proved to lie within their domains, and we corrected these by providing additional predicates to restrict the arguments.

As an alternative to formal proof, we do systematic inspection of specifications presented in tabular format. The following subsections present an extended example to illustrate the method and describe some errors we discovered.

5.1 Treatment motion controller

Here we consider one subsystem of our control program: the process that 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 [14, 16].

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 expected. 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.

5.2 Treatment motion controller formal description

The state of the 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:

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

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

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Controller}} \\
\hline
events : \mathbb{P}\, EVENT \\
clock, timer : TIME \\
status : STATUS \\
processing : PROCESSING \\
pending : \mathrm{seq}\, COMMAND \\
expected : \mathrm{seq}\, RESPONSE \\
\hline
\end{array}$$

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. No other state variables change when an event is signalled. This state transition is expressed by the *Signal* operation schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Signal}} \\
\hline
\Delta Controller \\
e? : EVENT \\
\hline
events' = events \cup \{e?\} \\
\ldots \\
\hline
\end{array}$$

The two basic controller operations are *Wait* and *HandleEvent*. The controller process executes *Wait* to wait for an event. The processing state switches from *run* to *wait*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Wait}} \\
\hline
\Delta Controller \\
\hline
processing = run \\
processing' = wait \\
\ldots \\
\hline
\end{array}$$

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.

$$
\begin{array}{|l}
\hline
\textit{HandleEvent} \\
\Delta \textit{Controller} \\
e? : \textit{EVENT} \\
\hline
e? \in \textit{events} \\
\textit{processing} = \textit{wait} \\
\textit{processing}' = \textit{run} \\
\textit{events}' = \textit{events} \setminus \{e?\} \\
\ldots \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\hline
\textit{WaitCmd} \\
\textit{Controller} \\
\hline
\textit{pending} = \langle\rangle \\
\textit{expected} = \langle\rangle \\
\hline
\end{array}
$$

After the user causes a command event to be signalled, *HandleCmd* loads the appropriate sequence of new commands. If the controller is not *ok* this handler is disabled, except for certain *restart* commands which are provided to recover from the error state.

$$
\begin{array}{|l}
\hline
\textit{HandleCmd} \\
\textit{HandleEvent} \\
c : \textit{COMMAND} \\
\hline
\textit{WaitCmd} \\
e? \in \textit{cmd} \\
\textit{status} = \textit{ok} \lor e? \in \textit{restart} \\
(\mathbf{let}\ \textit{cs} == \textit{commands}\ e? \bullet \\
\qquad c = \textit{head}\ \textit{cs} \land \textit{pending}' = \textit{tail}\ \textit{cs}) \\
\textit{status}' = \textit{status} \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Send}} \\
\hline
\Delta \textit{Controller} \\
c : \textit{COMMAND} \\
\textit{data}? : \textit{DATA} \\
\textit{message}! : \textit{MESSAGE} \\
\hline
\textit{message}! = \ldots \text{format msg from } c \text{ and } \textit{data}? \\
\textit{expected}' = \textit{responses}\ c \\
\hline
\end{array}
$$

These two operations are combined to send a new command

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

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

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

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ControllerError}} \\
\hline
\textit{HandleEvent} \\
\hline
\textit{state}' = \textit{error} \\
\textit{pending}' = \langle\rangle \\
\textit{expected}' = \langle\rangle \\
\hline
\end{array}
$$

5.3 Tabular presentation

We defined thirteen Z operations for this subsystem; each defines one state transition. To show how the operations work together we collect the pertinent formulas in one place. Table 3 is a state transition table, similar to the mode transition tables of the SCR notation [2].

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, 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 $\textit{processing} = \textit{run} \land \textit{WaitCmd}$.

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 3: Controller operations, preconditions, and progress postconditions

5.4 Analysis

Many properties of our controller specification can be confirmed by inspecting Table 3.

5.4.1 Generic properties

Generic properties such as completeness and determinism do not depend on the particular features of the application.

Completeness: The controller specification 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 $WaitCmd \lor WaitReceive$, etc.

Determinism: The controller specification 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.

5.4.2 Application-specific properties

These invariance and progress properties are particular to this application.

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 $WaitCmd \lor WaitReceive$ must be an invariant. The negation of the invariant is the forbidden state where an unsent command is pending but no reply is expected.

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.

5.5 Error discovery

Z specifications are built up from schemas spread over many pages, so it is easy to introduce errors and outright inconsistencies. For example:

Inconsistent includes: Schema C includes A and B, but A is inconsistent with B. Usually occurs because A was revised after C was written.

Displaced precondition: Operations B and C both include operation A. Precondition P should apply only to B but appears in A, so the precondition of C is too strong. Usually occurs because A was revised by splitting into a new A, plus B and C.

For example we discovered this 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*.

A simple remedy to these kinds of errors is to collect the predicates in one place (Table 3); this often reveals the errors. We observe that many of our schemas are inconsistent at first, perhaps as a result of our efforts to make predicates as strong as possible. A tool that could automatically identify obvious inconsistencies in expanded schemas would be quite useful, particularly if it could present its findings informatively, for example $x' = a \wedge x' = b$ or $x' = x \wedge x' = b$.

More subtle errors arise when collections of operations must work together. For example:

Omitted case: Sometimes occurs because the author was thinking of a poorly chosen (overly restricted) example. Common variations are **Missing operation** and **Precondition too strong**.

Overbooked state variable: Operation A stores a value in state variable x, intending it to be found by operation B. However operation C can store a different value in x before B is invoked.

Implicit invariant: The implicit invariant formed by the disjunction of the preconditions of all the operations is stronger than the explicit state invariant. However the postcondition of one operation does not satisfy this implicit invariant. After that operation executes, no operations are enabled and the system deadlocks.

For example we discovered these two missing operations: we forgot *RestartTimer*, and we only provided a single *Receive* operation instead of *ReceiveAck*, *ReceiveComplete* and *NextCommand*.

These errors cannot be detected by analyzing schemas one at time; it is necessary to consider whether the entire collection of operations satisfies some progress property. Our tabular inspection method provides one way to do this; Evans [5] proposes another method involving Unity-like proof rules. Model checking provides a third possibility.

5.6 An experiment in model checking

We translated the TMC specification presented here into the input language of the SMV temporal logic model checker. SMV is less expressive than Z so we had to create a simpler model that still represents the interesting behaviors. For example in Z we modelled pending commands and expected responses by sequences; in SMV we only modelled the lengths of the sequences. We checked the SMV specification against the application-specific invariance property $WaitCmd \lor WaitReceive$ (`AG (WaitCmd | WaitReceive)` in SMV) and the simplest possible progress property, that a waiting process will eventually run (`AG (!run -> AF run)`). The checker detected some seeded errors we had discovered during our inspections, but detected no errors in our lastest (corrected) version [13].

6 Other lessons

We offer some other observations based on our experience 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.
- 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.
- Inspection and simple paper-and-pencil analyses can detect most, but not all, errors in a large Z specification if it is sufficiently well partitioned to be reviewed in small sections.
- A formal description that contains minor errors can still be useful as a guide for coding and test planning.

Acknowledgments

The author thanks Ruedi Risler, Ira Kalet, Michael Patrick, Stan Brossard and Peter Wootton for contributions to the informal specification; Michael Patrick and Jonathan Unger for contributions to the formal specification and implementation guide; Michael Patrick, David Reid and Jonathan Unger for contributions to the code; and Mark Saaltink for running the domain check.

References

[1] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon Reese. Model checking large software specifications. In David Garlan, editor, *SIGSOFT '96: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–166, 1996. (also published as *ACM Software Engineering Notes* 21(6), Nov. 1996).

[2] 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.

[3] 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.

[4] L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal. EPICS architecture. In C. O. Pak, S. Kurokawa, and T. Katoh, editors, *Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 278–282, 1991. ICALEPCS, KEK, Tsukuba, Japan.

[5] Andy S. Evans. Specifying and verifying concurrent systems using Z. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, pages 366–380. Springer-Verlag, 1994. (Lecture Notes in Computer Science number 873).

[6] Thomas. W Griffin, Thomas F. Pajak, George E. Laramore, W. Duncan, M. P. Richter, F. R. Hendrickson, and Moshe H. Maor. Neutron vs photon irradiation of inoperable salivary gland tumors: Results of an RTOG-MRC cooperative randomized study. *International Journal of Radiation Oncology, Biology and Physics*, 15:1085–1090, 1988.

[7] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency analysis of state-based requirements. In *Proceedings of the 17th International Conference on Software Engineering*, pages 3 – 14, 1995.

[8] 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).

[9] 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.

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

[11] Jonathan Jacky. Safety-critical computing: Hazards, practices, standards, and regulation. In Rob Kling, editor, *Computerization and Controversy: Value Conflicts and Social Choices*. Academic Press, 1996. (second edition).

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

[13] Jonathan Jacky and Michael Patrick. Modelling, checking, and implementing a control program for a radiation therapy machine. In Rance Cleaveland and Daniel Jackson, editors, *AAS '97: Proceedings of the First ACM SIGPLAN Workshop on Automated Analysis of Software*, pages 25 – 32, 1997.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Jonathan Jacky, Jonathan Unger, Michael Patrick, David Reid, and Ruedi Risler. Experience with Z developing a control program for a radiation therapy machine. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 317 – 328. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science 1212.

[21] 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.

[22] 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), 1997.

[23] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

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

[25] Mark Saaltink. The Z/EVES system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 72 – 85. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science 1212.

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

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

[28] 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.