Formal Development of a Graphical User Interface for a Radiation Therapy Machine

Jonathan Jacky *
Jonathan Unger

Radiation Oncology Department RC-08
University of Washington
Seattle, WA 98195

Submitted to:
ZUM '95 Ninth International Conference of Z Users

November 23, 1994

Abstract

We wrote a formal specification in Z for the graphical user interface of a radiation therapy machine. We implemented our specification in a Pascal dialect on a workstation that uses the X window system to manage the keyboard and display. We initially model the user interface as a collection of separate Z operation schemas corresponding to sections in the informal prose requirements document. From these we derive a state machine model, represented as a state transition table whose entries are schema names from the Z specification. Our state transition table format compactly represents nested states that are modelled in Z by schema inclusion. We implement each table entry as a Pascal function or procedure. We also implement a dispatcher that selects the proper state transition whenever any X event occurs; our dispatcher is the X event loop. Our dispatcher is a table-driven interpreter that can handle any state transition system expressed in the format we defined. We model the dispatcher in Z and formally derive some of its code.

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

1 Introduction

The Clinical Neutron Therapy System at the University of Washington is a cyclotron and treatment facility that provides particle beams for cancer treatments with fast neutrons, production of medical isotopes, and physics experiments. This paper concerns the control program for the operator's console that therapists use to set up and deliver treatments to patients. The programmable part of the console is a workstation that runs a commercial real-time operating system [2] and uses the X window system to manage the keyboard and display [11, 12]. This console is just one component of a large control system that includes several computers and many non-programmable elements. The delegation of functions among the software and hardware components is described elsewhere [7, 8].

2 Why use a formal notation?

We already have a thorough description of our system in prose and pictures that the users consider to be complete [8]. Why go to all the effort of writing a second description of the same behaviors in Z [13]?

Safety issues motivate much of our formalization effort [4, 5, 6]. We want to show that our machine meets generic requirements for safety and completeness such as those proposed by Jaffe, et al [10].

In this paper we emphasize the use of Z as a detailed design notation and show how we derive program code from Z texts. We observe that the informal description of the user interface is repetitious; many operations work almost the same way. We see an opportunity to make the program small by factoring out common features and similar behaviors. This factoring out is only suggested by the prose description; in Z we make it explicit. The Z texts are no mere transliteration of the prose requirements. They are a different expression of the same behaviors, in a form that is more concise and better organized to serve as a guide for programming.

We formalize only those aspects of the the user interface which are pertinent to this design task. We do not attempt to treat "look and feel" aspects such as the appearance of the display. They are already described in sufficient detail in the informal requirements [8].

3 Informal requirements

The purpose of the treatment 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 many machine settings that must be set properly to deliver each prescribed treatment. The console program enables the operator to choose patients and fields from the prescription database, and ensures that the radiation beam can only turn on (through a separate nonprogrammable subsystem) when the prescribed settings for the chosen field have been achieved.

The informal requirements for the console program, including but not limited to its user interface, comprise 45 pages of prose and diagrams (chapter 8 in [8]) which describe the activities associated with about a dozen different screens. For example, Fig. 1 shows the fields available for the currently selected patient, and Fig. 2 shows information about the two dosimeters (which are replicated for safety) as well as some safety interlocks. Operators select these screens by pressing dedicated function keys, and can also position a cursor over particular items on each screen and select them (for example to choose one of the fields shown in Fig. 1 in order to load its prescribed settings). Operators can also enter or modify values for some items after they select them, by typing at the workstation keyboard.

4 Formal model

A 17 page section in [9] provides a formal definition for the user interface to every operation described in the prose requirements [8]. This section includes about 450 lines of Z text (expressed as LaTeXsource) divided between about 50 schemas and some axiomatic definitions. The following summary is simplified for brevity.

Each operation from the prose is modelled by one or more Z operation schemas on the *Console* state. The *display* state variable indicates which of the screen designs (such as those shown in Figs 1 and 2) is currently visible. The *edit* state variable indicates what kind of user interaction is currently in progress: it is *idle* when no interaction is in progress and the console is waiting for input, *editing* when the user is entering or modifying a value and the console is waiting for the user to type a character, etc. The *item* state variable indicates the name (not the value) of the item which is being modified, while *buffer* models the (possibly incomplete) string that the user edits. Most operations are only *Available* when editing is not already in progress; otherwise, the console is *Engaged*.

$$EDIT ::= idle \mid editing \mid \ldots$$

OPERATOR: Adam Smith PATIENT: Test Patient FIELD: None

GANTRY /PSA	FILTER /WEDGE	LEAF COLLIM	DOSI- METRY	THERAPY INTLKS	PROTON BEAM

FIELDS

	Field	Fractions	To date	MU	Total	Expected	To date
1	Anterior	16	12	122	1952	1464	1464
2	Left Lateral	16	12	139	2224	1668	1668
3	Posterior	16	11	124	1984	1364	1364
4	Right Lateral	16	11	135	2160	1485	$\boxed{1413}$

Figure 1: Fields display

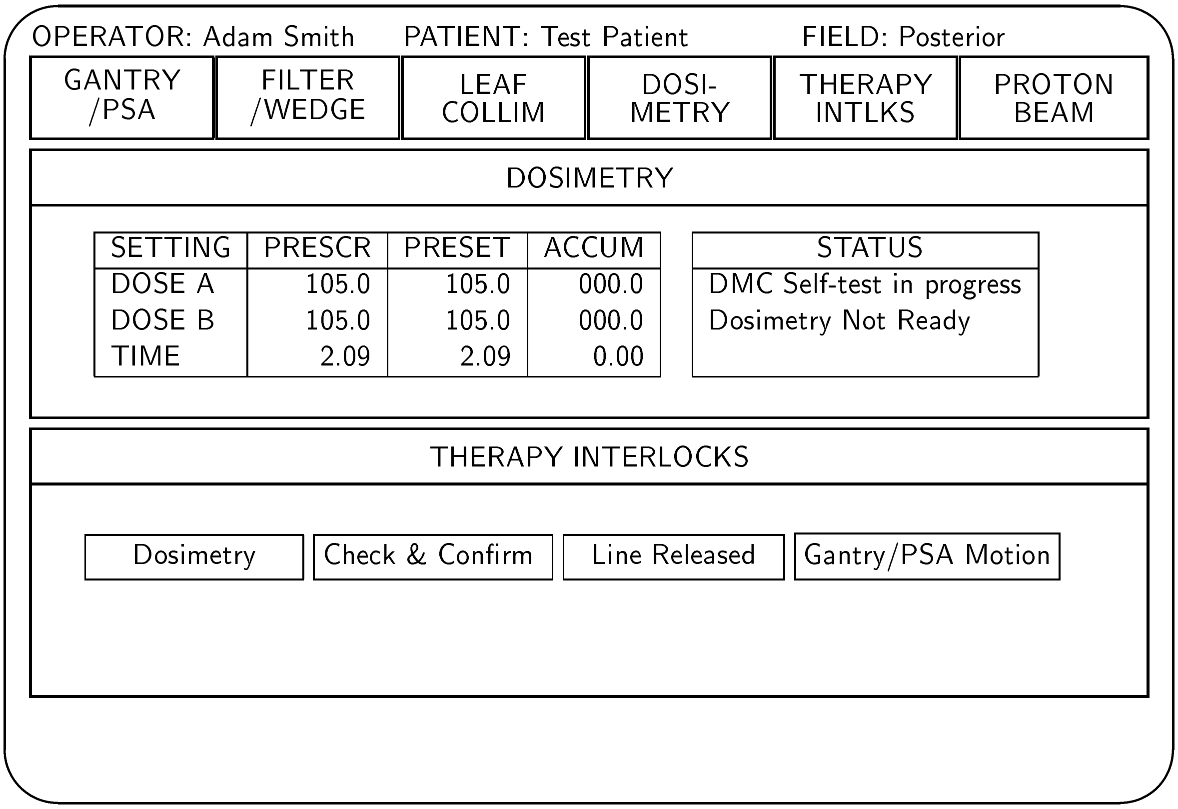

Figure 2: Dosimetry and interlocks display

$$
\begin{array}{|l}
\hline
\textit{Console} \\
\textit{display} : \textit{DISPLAY} \\
\textit{edit} : \textit{EDIT} \\
\textit{item} : \textit{NAME} \\
\textit{buffer} : \textit{STRING} \\
\\
\ldots \\
\hline
\end{array}
$$

$$
\begin{aligned}
\textit{Available} &\mathrel{\widehat=} [\,\textit{Console} \mid \textit{edit} = \textit{idle}\,] \\
\textit{Engaged} &\mathrel{\widehat=} [\,\textit{Console} \mid \textit{edit} \in \{\,\textit{editing}, \ldots\,\}\,]
\end{aligned}
$$

Operations occur whenever the user provides input at the workstation by typing or pressing a function key (we do not use the workstation "mouse"). Every input event is modelled by the *Event* operation schema. Each value of $INPUT$ represents a different X event, but we can ignore many of them.

$$
\begin{array}{|l}
\hline
\textit{Event} \\
\Delta \textit{Console} \\
\textit{input}? : \textit{INPUT} \\
\hline
\end{array}
$$

$$
\textit{Ignore} \mathrel{\widehat=} \textit{Event} \wedge \Xi \textit{Console}
$$

When the console is $Available$, the user may select a new *display* by pressing a function key (for example to choose one of the displays shown in Figs. 1 or 2, among others). We need a function *name* to associate the *input*? (the function key that the user presses) with the name of the $DISPLAY$ that the user wants to see. The newly selected $display'$ appears and console remains $Available$. Here we do not need to model the details of updating the display contents.

$$
\begin{array}{|l}
\hline
\textit{SelectDisplay} \\
\textit{Event} \\
\hline
\textit{Available} \\
\textit{Available}' \\
\textit{name}\ \textit{input}? \in \textit{DISPLAY} \\
\textit{display}' = \textit{name}\ \textit{input}? \\
\hline
\end{array}
$$

The *SelectDisplay* operation schema corresponds to a single operation in the informal prose description. Other operations from the informal description must be modelled as several Z operations. Each editing operation is modelled as at least three Z operations: one to begin *Edit*ing (when the user presses an appropriate function key), another to *Get* each keystroke and modify the buffer (usually just by adding the new character to the end), and a third to *Accept* the new value when editing is finished (signalled when the user types a terminator character, such as the RETURN key).

$$
\begin{array}{l}
\textbf{Edit} \\
\hline
\mathit{Event} \\
\hline
\mathit{Available} \\
\mathit{Engaged}' \\
\mathit{buffer}' = \mathit{empty} \\
\mathit{display}' = \mathit{display} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{Get} \\
\hline
\mathit{Event} \\
\hline
\mathit{Engaged} \\
\mathit{Engaged}' \\
\mathit{buffer}' = \mathit{modify}\ \mathit{buffer}\ \mathit{input?} \\
\mathit{display}' = \mathit{display} \\
\mathit{item}' = \mathit{item} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{Accept} \\
\hline
\mathit{Event} \\
\hline
\mathit{Engaged} \\
\mathit{Available}' \\
\mathit{input?} \in \mathit{terminator} \\
\mathit{buffer}' = \mathit{buffer} \\
\mathit{display}' = \mathit{display} \\
\hline
\end{array}
$$

Edit and *Accept* are building blocks; we specialize them to describe particular editing operations. The simplest example occurs when the operator types a message to annotate an event log (which is mostly written automatically). The user presses the WRITE LOG MESSAGE function key to invoke the *EditMessage* operation, a specialization of *Edit*. This results in the *EditingMessage* state:

$$
\begin{array}{l}
\textbf{EditMessage} \\
\hline
\mathit{Edit} \\
\hline
\mathit{input?} = \mathit{log\_message} \\
\mathit{item}' = \mathit{name}\ \mathit{log\_message} \\
\hline
\end{array}
$$

$$EditingMessage \mathrel{\widehat=} [\, Engaged \mid item = name\ log\_message \,]$$

As the user types, *Get* collects characters in *buffer*. When the user types a terminator character, the *WriteMessage* operation, a specialization of *Accept*, writes the completed *message*! to the event log.

$$\begin{array}{|l}
\hline
\textit{WriteMessage} \\
\hline
Accept \\
message! : STRING \\
\hline
EditingMessage \\
message! = buffer \\
\hline
\end{array}$$

Most editing operations work in a similar fashion, but the values that they collect change the underlying machine state instead of merely appearing in a file.

5 Combining the operations

The Z texts in Section 4 do not describe any explicit control structure that invokes operations when they are requested. We must design and implement this control structure. We begin by combining all top-level operations into a single *ConsoleOp* schema (not including building blocks such as *Event*, *Edit* and *Accept* which are only used to define other operations).

$$ConsoleOp \mathrel{\widehat=} SelectDisplay \lor EditMessage \lor WriteMessage \lor \ldots \lor IgnoreOthers$$

This *ConsoleOp* operation is invoked whenever the user provides any input (keypress). The control structure is implicit in the preconditions of all the constituent operations. We have tried to define each of these so that the precondition of exactly one is satisfied each time *ConsoleOp* is invoked. *IgnoreOthers* is the default do-nothing operation whose precondition is the negation of the disjunction of the preconditions of all the other operations.

6 State transition table

The *ConsoleOp* operation defines a *finite-state machine* where each of the constituent operation schemas describes a single transition. (Any state machine constructed from Z operation schemas in this way is a *Mealy machine* whose outputs are associated with state transitions. The analyses of Jaffe, et al. [10] are based on a Mealy model.) Table 1 shows the state transition table derived from the Z texts in [9] (which are more complicated than the simplified excerpts presented in Section 4). We developed code that can interpret any state machine

State	Input	Operation
$key = locked$	(all inputs)	*Locked*
Available	$name\ input? \in DISPLAY$	*SelectDisplay*
	$input? = log\_message$	*TypeMessage*
$run = running$	$input? = cancel\_run$	*SelectCancelRun*
Setup	$input? = auto\_setup$	*AutoSetup*
	$input? = expt\_mode$	*ExptMode*
	$input? = store\_field$	*EditField*
	$input? = edit\_setting$	*SelectSetting*
	$input? = override\_cmd$	*SelectOverride*
	$input? = logout$	*Logout*
ListingPatients	$name\ input? \in \mathrm{ran}\ patients$	*SelectPatient*
ListingFields	$name\ input? \in \mathrm{dom}\ fields$	*SelectField*
	(other inputs)	*Ignore*
Engaged	$input? \in CHAR$	*Get*
$\neg\ LoggedOut$	$input? = cancel$	*Cancel*
EditingCancel	$input? = confirm$	*CancelRun*
TypingMessage	$input? \in terminator$	*WriteMessage*
EditingField	$input? \in terminator$	*StoreField*
EditingSetting	$input? \in terminator$	*StoreSetting*
Overriding	$input? = confirm$	*Override*
LoggedOut	$input? \in terminator$	*Login*
	$input? = cancel$	*NoCancel*
	(other inputs)	*Ignore*

Table 1: User interface state transition table.

that is expressed in this tabular form, not just the particular state machine defined by the operations in [9]).

This table-driven interpreter (or dispatcher) is our alternative to the usual way to implement event-driven X programs, which is to code a a large `case` statement with a case branch for each event [11].[1] Our method has advantages when many of the events must be handled in a similar, but not exactly identical, fashion. In such cases our method results in code that is shorter. We also feel it is clearer (and should be easier to modify) because the factoring-out of similar features that appears in the Z texts is expressed in exactly the same way in the program code.

We build Table 1 by listing all of the operation schemas that are combined in $ConsoleOp$. These determine the contents (but not the order) of the third column. Then we extract the preconditions of each. We determined the preconditions by inspection, not by performing the Z precondition calculation described in [13]. Preconditions on $input?$ are listed in the middle column, while those involving other state variables are listed in the first column. In our table, sequence order and indentation represent the nesting of states that is expressed in Z by schema inclusion.

The precondition states in the first column are indicated by the names of the Z state schemas from [9] where they are defined. Solid double lines separate the mutually exclusive states $key = locked$, $Available$ and $Engaged$. Solid single lines, together with indentation, indicate that the table entries enclosed between them are *substates* of preceding entries. The full precondition of a substate is formed by conjoining the preconditions of the preceding entries at lesser indentation. For example, the full precondition of the substate on the row with $ListingPatients$ in the first column is $Available \land Setup \land ListingPatients$. This conjunction is the precondition given in [9] for the $SelectPatientC$ operation, that appears in the third column of the same row. When the first column in a row is blank, the substate is the same as the last preceding nonblank column: the precondition for the $Logout$ operation is $Available \land Setup$.

Dotted lines separate mutually exclusive substates. Thus $run = running$ and $Setup$ indicate mutually exclusive substates of $Available$, and $ListingPatients$ and $ListingFields$ indicate mutually exclusive substates of $Setup$.

The inputs in the middle column are expressed as predicates on the variable $input?$. The right column lists the operation schemas that define the next state, as well as any outputs. Each line also applies to any included substates, so $name\ input? \in DISPLAY$ elicits the $SelectDisplay$ operation in the $Available$ state, and also in its substates $run = running$, $Setup$, and in its sub-substates $ListingPatients$ and $ListingFields$.

[1] Bowen [1] modelled some display aspects of the X window system in Z, but did not consider event handling.

Our table suggests an efficient implementation. It should only be necessary to test each distinct state precondition once. It should not be necessary to test preconditions of substates that are indented under states whose preconditions have been found to be false.

7 Formalizing the state transition table

To develop a program that interprets the state transition table, we must express its meaning formally. Each entry in the table is described by the *Transition* schema. The indentation of each row is indicated by an integer nesting *depth*. The *state* and *input* preconditions in the first and second columns are indicated by predicates expressed as unary relations on *Console* and *INPUT*, respectively. The *operations* in the third column of the table are modelled by functions on *Console* states. Table 1 is modelled by t, a sequence of these records.

Transition

$$
\begin{array}{l}
depth : \mathbb{N} \\
state : \mathbb{P}\ Console \\
in : \mathbb{P}\ INPUT \\
op : Console \nrightarrow Console
\end{array}
$$

$$t : \mathrm{seq}\ Transition$$

Here are some excerpts from Table 1 in Z syntax. The entry for $t\,6$ shows how we model rows in Table 1 where the first column is blank: we set the *state* in the *Transition* record to the empty set, $\varnothing$.

$$
\begin{array}{l}
(t\,1).depth = 0 \\
(t\,1).state = \{\ Console \mid key = locked \bullet \theta Console\ \} \\
(t\,1).in = \{\ input? : INPUT \mid true\ \} \\
(t\,1).op = \{\ Locked \bullet (\theta Console, \theta Console')\ \}
\end{array}
$$

$$
\begin{array}{l}
(t\,5).depth = 1 \\
(t\,5).state = \{\ Setup \bullet \theta Console\ \} \\
(t\,5).in = \{\ input? : INPUT \mid input? = auto\_setup\ \} \\
(t\,5).op = \{\ AutoSetupC \bullet (\theta Console, \theta Console')\ \}
\end{array}
$$

$$
\begin{array}{l}
(t\,6).depth = 1 \\
(t\,6).state = \varnothing \\
(t\,6).in = \{\ input? : INPUT \mid input? = expt\_mode\ \} \\
(t\,6).op = \{\ ExptModeC \bullet (\theta Console, \theta Console')\ \}
\end{array}
$$

8 Interpreting the state transition table

In this section we formally define the core of interpreter: the *transition* function that takes any *Console* state and *input*? into a new *Console* state. The table t is a parameter of this function.

We interpret the table by traversing it from top to bottom, searching for an entry which is *enabled*. We say an entry is enabled if its full state precondition is satisfied: this is true when the precondition given in the first column of the entry is satisfied, and all the nearest preceding entries at lesser indentations are enabled as well. If the first column entry is blank, the entry is enabled if the immediately preceding entry is enabled.

We define a unary prefix relation e on table entries; $e(i)$ is *true* if table entry i is enabled. The unary prefix relation $edd(i)$ describes the effect of sequence order and indentation; it is *true* if all the nearest preceding entries at lesser indentation are enabled. For $e(i)$ to be *true*, $edd(i)$ must be *true*. Its definition uses the binary prefix relation $ed(i,d)$, which is *true* when the nearest entry indented at depth d that precedes entry i is enabled, or if there is no such preceding entry.

$$
\begin{array}{|l}
e\_, edd\_ : \mathbb{P}(\mathrm{dom}\, t) \\
ed\_ : \mathrm{dom}\, t \leftrightarrow \mathbb{N} \\
\hline
\forall i : \mathrm{dom}\, t \bullet \\
\qquad (\forall d : \mathbb{N} \bullet (\mathbf{let}\ ds == \{ j : 1 \mathinner{..} i-1 \mid (t\, j).depth = d \} \bullet \\
\qquad\qquad ed(i,d) \Leftrightarrow e(max\ ds) \lor ds = \varnothing)) \land \\
\qquad (\mathbf{let}\ d == (t\ i).depth \bullet \\
\qquad\qquad edd(i) \Leftrightarrow (\forall dd : 0 \mathinner{..} d-1 \bullet ed(i, dd))) \land \\
\qquad (\forall c : Console \bullet \\
\qquad\qquad e(i) \Leftrightarrow edd(i) \land (((t\ i).state = \varnothing \land e(i-1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \lor ((t\ i).state \neq \varnothing \land c \in (t\ i).state)))
\end{array}
$$

When we have found an enabled entry, we test whether the input precondition given in the second column is also satisfied. If it is, we say the transition has *triggered*. We then apply the function given in the third column to obtain the new state. If no entries are triggered, the new state is the same as the old state.

$$
\begin{array}{|l}
transition : (Console \times INPUT) \longrightarrow Console \\
\hline
\forall input? : INPUT;\ c : Console \bullet \\
\qquad transition(c, input?) = \\
\qquad\qquad \mathbf{if}\ \exists i : \mathrm{dom}\, t \bullet e(i) \land input? \in (t\ i).in\ \mathbf{then}\ (t\ i).op\ c\ \mathbf{else}\ c
\end{array}
$$

A single application of the *transition* function is the processing of a single input event. It is expressed by the *ConsoleOp*1 operation:

$$ConsoleOp1 \mathrel{\widehat=} [\, Event \mid \theta Console' = transition(\theta Console, input?) \,]$$

If the table t is constructed properly, this operation is the same as the *ConsoleOp* we defined in section 5. In other words, *ConsoleOp*1 is a *refinement* of *ConsoleOp*.

The refinement is correct because each entry in t corresponds to one disjunct in *ConsoleOp*. The only subtle point involves the preconditions of the substates (the indented table entries). The precondition of a substate is the conjunction of the condition listed in the substate's own table entry, and all the conditions listed in the nearest preceding entries at lesser indentation. We have expressed this formally by including *edd* in the definition of e.

9 Simplifying the interpreter

The formal definition of *transition* expresses our intent, but is not ready for translation to an executable program. The definition of e includes predicates about sets and quantifiers. Implementing these directly would result in a complicated, inefficient program. We can eliminate them.

The definition of e says that all nearest preceding entries at lesser indentation must be enabled. This holds only if *no* such entry is *not* enabled. We only need to keep track of a single preceding disabled entry: the one which is least indented. We record its depth of indentation *dd*.

When traversing the table, we may encounter sequences of disabled entries, where indented entries are disabled because preceding entries at lesser depth are also disabled. Our *dd* is the depth of the entry at the beginning of the most recently encountered sequence of disabled entries.

It turns out to be more convenient to keep track of all the preceding entries, not just those which are at a lesser depth than the current entry. The depth of the preceding entry *dp* might be greater than the depth of the current entry d. The depths of all the preceding disabled entries form the set *dds*, and *dd* is the smallest element in the set. If the depth of the current table entry d is not greater than *dd*, or if there are no preceding disabled entries (the set *dds* is empty), then the current entry can be enabled. We can express this argument formally:

$$
\begin{array}{ll}
\forall\, i : \mathrm{dom}\, t \bullet (\mathbf{let}\ d == (t\ i).depth;\ dp == (t\,(i-1)).depth \bullet & [\text{Define } d,\ dp] \\
\quad e(i) \Rightarrow edd(i) & [\text{Definition of } e(i)] \\
\qquad \Leftrightarrow (\forall\, dd : 0\,..\,d-1 \bullet ed(i, dd)) & [\text{Definition of } edd(i)] \\
\qquad \Leftrightarrow \neg\, (\exists\, dd : 0\,..\,d-1 \bullet \neg\, ed(i, dd)) & [\text{Predicate calculus}] \\
\qquad \Leftrightarrow (\mathbf{let}\ dds == \{\, dd : 0\,..\,dp \mid \neg\, ed(i, dd)\,\} \bullet & [\text{Define } dds] \\
\qquad\quad (0\,..\,d-1) \cap dds = \varnothing & [\text{Definitions of ranges for } ed,\ dds] \\
\qquad\quad \Leftrightarrow dds = \varnothing \vee d - 1 < min\ dds & [\text{Set theory, range arithmetic}] \\
\qquad\quad \Leftrightarrow dds = \varnothing \vee d \leq min\ dds & [\text{Arithmetic, } < \text{ vs. } \leq] \\
\qquad\quad \Leftrightarrow (\mathbf{let}\ dd == min\ dds \bullet dds = \varnothing \vee d \leq dd))) & [\text{Define } dd]
\end{array}
$$

This result is useful because we can easily keep track of the existence of dds and the value of dd without maintaining an explicit representation of the whole set dds. The only predicate we actually need to test in the implementation is the final one:

$$dds = \varnothing \vee d \leq dd$$

The $dds = \varnothing$ condition occurs when we are not traversing a sequence of disabled entries, and the $d \leq dd$ condition occurs when we have just emerged from the end of such a sequence.

10 Implementing the interpreter

Our implementation language is a (nonstandard) Pascal dialect [3] that provides pointers to functions and procedures. This makes it easy to translate t and *Transition*, the declarations for the state transition table, from Z.[2]

```
type
    Transition = record
        depth:  integer;
        state:  ^function;
        in:  ^function;
        op:  ^procedure;
    end;

var
    t:  array[0..n] of Transition;
```

[2]Actually, the syntax for pointing to functions and procedures is a bit more complicated than indicated here.

The `state` and `in` functions are implemented in Pascal as boolean functions that test the console state variables and the input, respectively, and return `true` if the preconditions are satisfied. Table entries whose first column is blank (as in the entries following *Setup*) are indicated by assigning the `state` pointer to `nil`. The `op` procedures are implemented as Pascal procedures that perform the state changes (assignment statements) called for in the operation schemas. Our `op` procedures do not check any preconditions, but rely on the `state` and `in` functions for this.

Our implementation of the *transition* function is presented here as a Pascal procedure without parameters; the table t, the *input*? and the *Console* state variables are all global. Its local variables derive from the formal development.

```
function transition;

var
    i:  integer;       { i              index of current table entry                    }
    e:  boolean;       { e(i)           current entry is enabled                        }
    ed:  boolean;      { e(i - 1)       preceding entry is enabled                      }
    empty:  boolean;   { dds = ∅        no preceding disabled entries                   }
    d:  integer;       { d              (t i).depth, depth of current table entry       }
    dd:  integer;      { dd             depth of least indented preceding disabled entry }
    tr:  boolean;      { e(i) ∧         current entry is triggered                      }
                       { input? ∈ (t i).in                                              }
```

Our `transition` code tests each table entry in turn. Our Pascal dialect [3] provides a (non-standard) `invoke` function that executes a function or procedure indicated by a pointer.[3] This makes it easy to translate the formal definitions of e and *transition* from section 8. We implemenent $p \Leftrightarrow q \wedge r$ as `if q then p := r else p := false` instead of `p := q and r` in order to avoid the expense of evaluating r when q is *false*.

[3]The actual syntax of `invoke` is a bit more complicated than shown here.

```
function transition;
    ...
    while i < n ...
        ...
        { Invariant:   dds = ∅ ∨ dd = min dds  ...}
        if empty or (d <= dd) then
            if (t[i].state = nil)
                then e := ed;
                else e := invoke(t[i].state)
            else e := false;

        if e then tr := invoke(t[i].in) else tr := false;
        if tr then invoke(t[i].op);

        { Maintain invariant }
        ...
```

This code is correct if the program variables satisfy their formal definitions when the assignment to `e` is made. This condition is the *loop invariant*: it must be true each time execution reaches the first `if` statement inside the `while` loop. We first establish the invariant by assigning values to program variables before entering the loop. Then, each time `e` is assigned in the body of the loop, new values must also be assigned to other variables in order to maintain the invariant. The only tricky part of the invariant is $dds = \emptyset \vee dd = min\, dds$.

To maintain this invariant, three conditions are pertinent: $e(i)$, $dds = \emptyset$, and $d \leq dd$. In the following discussion the unprimed variables dd and dds represent the values at the top of the loop when `e` is assigned, and the primed variables dd' and dds' represent the new values assigned at the bottom of the loop, that will apply when `e` is assigned in the next turn of the loop. The correct assignments to dd' and $dds' = \emptyset$ can be determined by a case analysis, shown in Table 2. A formal justification of the case analysis appears in Appendix A.

We only need to write code for cases where the new (primed) values dd' or dds' differ from their initial (unprimed) values. The analysis reveals three such cases, which are highlighted by boxes in the table: when a disabled entry is encountered after a sequence of enabled entries (case 5), when an enabled entry is encountered after a sequence of disabled entries (case 8), and when a disabled entry at lesser depth is encountered after a sequence of disabled entries at greater depth (case 7). Writing out the code for each case, we obtain

$e(i)$	$dds = \varnothing$	$d \leq dd$	$dds' = \varnothing$	dd'
false	*false*	*false*	*false* (2)	dd (3)
true	*false*	*false*	XX (4)	XX (4)
false	*true*	X (1)	$\boxed{false}$ (2,5)	$\boxed{d}$ (5)
true	*true*	X (1)	*true* (6)	X (1)
false	*false*	*true*	*false* (2)	$\boxed{d}$ (7)
true	*false*	*true*	$\boxed{true}$ (8)	X (1)

Table 2: Maintaining the invariant $dds = \varnothing \vee dd = min\ dds$

```
{ Maintain invariant }
if (not e) and empty then begin empty := false; dd := d end; { case (5) }
if (not e) and (not empty) and (d <= dd) then dd := d;       { case (7) }
if e and (not empty) and (d <= dd) then empty := true;       { case (8) }
```

We only need to add a little loop machinery to complete the development. The completed `transition` function appears in Fig. 3. The complete dispatcher is simply a loop that, on each turn, removes one event from the head of the X event queue and calls `transition`. Almost all of the work in our application is done by code that lies outside the X event loop: `transition` itself, and the `state`, `in` and `op` functions and procedures indicated by the pointers in table `t`, that actually implement the schemas from [9]. This differs from the style of implementing X programs recommended in [11], where the event loop contains a large `case` statement with a case branch for each event.

In our implementation the dispatcher, including the `transition` function, the rest of the X event loop code, and table `t` comprise about n lines of Pascal code (including plenty of comments and blank lines). The functions and procedures that implement the schemas comprise another m lines.

References

[1] Jonathan P. Bowen. X: Why Z? *Computer Graphics Forum*, 11(4):221–234, October 1992.

[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. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 388–402, Odense, Denmark, April 1993. First International Symposium of Formal Methods Europe, Springer-Verlag. Lecture Notes in Computer Science 670.

[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, 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.

[9] Jonathan Jacky and Jonathan Unger. Formal specification of control software for a radiation therapy machine. Technical Report 94-07-01, Radiation Oncology Department, University of Washington, Seattle, WA, July 1994.

[10] Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

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

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

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

```
function transition;

var
    i:  integer;        { i             index of current table entry                     }
    e:  boolean;        { e(i)          current entry is enabled                         }
    ed:  boolean;       { e(i − 1)      preceding entry is enabled                       }
    empty:  boolean;    { dds = ∅       no preceding disabled entries                    }
    d:  integer;        { d             (t i).depth, depth of current table entry        }
    dd:  integer;       { dd            depth of least indented preceding disabled entry }
    dleq:  integer;     { d ≤ dd        beginning new sequence of disabled entries       }
    tr:  boolean;       { e(i) ∧        current entry is triggered                       }
                        { input? ∈ (t i).in                                              }

begin
    i := 0; ed := true; tr := false; empty := true;
    while (i < n) and (not tr) do
    begin
        d := t[i].depth
        dleq := d <= dd;

        { Invariant:   dds = ∅ ∨ dd = min dds  }
        if empty or dleq then
            if (t[i].state = nil)
                then e := ed;
                else e := invoke(t[i].state)
            else e := false;

        if e then tr := invoke(t[i].in) else tr := false;
        if tr then invoke(t[i].op);

        { Maintain invariant  }
        if (not e) and empty then begin empty := false; dd := d end;
        if (not e) and (not empty) and dleq then dd := d;
        if e and (not empty) and dleq then empty := true;

        ed := e;
        i := i + 1;
    end;
end;
```

Figure 3: Completed `transition` function.

A Maintaining the invariant

This section presents the justifications for the numbered entries in Table 2. Before working through the table, we establish some generally useful results. It is convenient to package up pertinent state variables, definitions and the loop invariant in the *Loop* schema. The *Step* operation schema models the effect of executing the code at the end of the loop in Fig 3 (after the comment, "Maintain invariant"). Note that the primed "after" variables are defined automatically for us by the usual Z convention.

$$
\begin{array}{|l}
\textit{Loop} \\
\hline
i : \mathrm{dom}\, t \\
d, dd : \mathbb{N} \\
dds : \mathbb{P}\, \mathbb{N} \\
\hline
d = (t\ i).depth \\
dds = \{\, dd : 0 \mathinner{..} (t\,(i-1)).depth \mid \neg\ ed(i, dd) \,\} \\
dds = \varnothing \lor dd = min\ dds \\
\hline
\end{array}
$$

$$Step \mathrel{\widehat{=}} [\, \Delta Loop \mid i' = i + 1 \,]$$

We consider all the formal text that follows to be within the scope of the definitions in *Step*. Now we can derive some results.

A. The range of dds' is $0 \mathinner{..} d$.

$$
\begin{array}{ll}
max\ dds' \leq (t\,(i'-1)).depth & [\text{Def'n of } dds \text{ range, }' \text{ naming convention}] \\
\quad = (t\,((i+1)-1)).depth & [i' = i + 1] \\
\quad = (t\ i).depth & [\text{Arithmetic}] \\
\quad = d & [d = (t\ i).depth]
\end{array}
$$

B. The enabled state of the current entry becomes the enabled state of of the most recent entry at its own depth: $e(i) \Leftrightarrow ed(i', d)$

$$
\begin{array}{ll}
(\textbf{let}\ ds' == \{\, j : 1 \mathinner{..} i'-1 \mid (t\,j).depth = d \,\} \bullet & [\text{Def'n of } ed(i,d) \text{ in section 8}] \\
e(i) \Leftrightarrow e(max\ ds') & [i'-1 = i,\ i = max\, 1 \mathinner{..} i, \text{ def'n of } d] \\
\quad \Leftrightarrow ed(i, d')) & [\text{Definitions of } e(i),\ ed(i,d)]
\end{array}
$$

C. When an entry is not enabled, its depth becomes one of the preceding disabled depths: $\neg\ e(i) \Leftrightarrow d \in dds'$.

$$
\begin{array}{ll}
\neg\ e(i) \Leftrightarrow \neg\ ed(i', d) & [\text{Result B}] \\
\quad \Leftrightarrow d \in dds' & [\text{Result A, definitions of } d,\ dds']
\end{array}
$$

D. When an entry is enabled, for subsequent entries the set of preceding disabled entries is empty: $e(i) \Leftrightarrow dds' = \varnothing$.

$$\begin{aligned}
e(i) &\Rightarrow edd(i) && \text{[Definition of } e(i)] \\
&\Leftrightarrow (\forall\, dd : 0 \mathrel{..} d-1 \bullet ed(i, dd)) && \text{[Definition of } edd(i)] \\
&\Leftrightarrow (0 \mathrel{..} d-1) \cap dds' = \varnothing && \text{[Definition of } dds'] \\
&\Leftrightarrow dds' = \varnothing && \text{[A: } dds' \subseteq 0 \mathrel{..} d\text{, but C: } d \notin dds']
\end{aligned}$$

Now we can work through the entries in Table 2.

1. When $dds = \varnothing$ the value of dd doesn't matter. The invariant holds because

   $$dds = \varnothing \;\Rightarrow\; (dds = \varnothing \vee dd = min\ dds) \qquad [p \Rightarrow p \vee q]$$

   X indicates this "don't care" condition. We only need six rows in the table, not eight.

2. When an entry is disabled, its own depth must be one of the elements of dds', which cannot be empty.

   $$\neg\, e(i) \Rightarrow dds' \neq \varnothing \qquad \text{[C: } d \in dds']$$

3. When an entry is disabled but its own depth is greater than dd, then dd remains the smallest element in dds'.

   $$\begin{aligned}
   &\neg\, e(i) \wedge dds \neq \varnothing \wedge d > dd && \text{[Entry in Table 2]} \\
   &\quad\Rightarrow dds' = dds \cup \{d\} && \text{[C: } d \in dds'\text{, definition of } dds'] \\
   &\quad\Rightarrow dd = min\ dds' && [dd < d] \\
   &\quad\Leftrightarrow dd' = dd && \text{[Definition of } dd']
   \end{aligned}$$

4. Here the entry is enabled but its depth is greater than dd. This condition is impossible because it contradicts the result about $e(i)$ we derived in section 9.

   $$\begin{aligned}
   &e(i) \wedge dds \neq \varnothing \wedge d > dd && \text{[Entry in Table 2]} \\
   &\quad\Leftrightarrow \mathit{false} && \text{[Section 9: } e(i) \Rightarrow dds = \varnothing \vee d \leq dd]
   \end{aligned}$$

   Inspection of the code confirms that this condition is prevented by the `if` ... `then` ... `else` ... statement in the body of the loop. XX indicates this impossible condition.

5. A disabled entry is first encountered when dds is empty. The depth of this entry becomes the first — and necessarily the smallest — element in dds'

   $$\neg\, e(i) \wedge dds = \varnothing \qquad \text{[Entry in Table 2]}$$

$$\Leftrightarrow dds' = \{d\} \qquad \text{[C: } d \in dds'\text{]}$$
$$\Rightarrow dds' \neq \varnothing \wedge d = min\ dds' \qquad \text{[Set theory, definition of } min\text{]}$$
$$\Leftrightarrow dd' = d \qquad \text{[Definition of } dd'\text{]}$$

6. If the entry is enabled and dds is empty, it remains so; dds' is empty, by result D.

7. This disabled entry lies at lesser indentation depth than any nearest preceding disabled entry, so dd must be reassigned to maintain the invariant.

$$\neg\, e(i) \wedge d \leq dd \qquad \text{[Entry in Table 2]}$$
$$\Rightarrow d = min\ dds' \qquad \text{[C: } d \in dds',\ d \leq min\ dds\text{]}$$
$$\Leftrightarrow dd' = d \qquad \text{[Definition of } dd'\text{]}$$

8. An enabled entry follows a disabled entry that was indented to an equal or greater depth. This signals the end of a sequence of nested disabled entries. By result D, dds' should be emptied.

This concludes the case analysis.