Formal Specification and Development of Control System Input/Output

Jonathan Jacky

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

jon@radonc.washington.edu

Technical Report 92-05-02

May, 1992

Abstract

This report presents a formal specification in the Z notation for computations that calculate control system state variables from input/output device register contents (and vice-versa). The specification is motivated by a particular medical device but is quite generic and should be widely applicable. The specification is parameterized so that an implementation can be adapted to different control systems by providing tables of configuration data, rather than changing executable code. Specified behaviors include detection of errors (where clients invoke operations with invalid parameters) and faults (where input/output devices report invalid data). The specification is not merely descriptive, but is also used in the formal development (or "refinement") of a detailed design. From an initial specification which naturally expresses the requirements, but is abstract and non-constructive, we derive a functionally equivalent specification (also in Z), which suggests a straightforward and efficient implementation in an imperative programming language. Formal justification is provided for each step in the derivation. Conjectures are posed that formalize claims such as "All inputs are handled properly." Proving the conjectures could check for errors in the derivation, and provide confidence that the formal specification expresses the intended requirements.

1 Introduction

Safety-critical control systems are often advocated as ideal applications for formal software development methods [2]. However, few published case studies apply formal methods to a problem of central importance in these applications: low level input/output[1]. Control systems often include large numbers of input and output signals, and devote much computational effort to dealing with them. The system that motivates this study, a cyclotron used for medical isotope production and radiation therapy for cancer, has more than one thousand input and output signals [5, 7].

A vital activity in most control systems is the processing of unscaled, unencoded contents of the input/output device registers that are attached to the transducers that sense and control some physical process. Register contents must be translated to the values of the state variables that occur in the control laws that the system is supposed to obey. For example, analog quantities must be scaled and offset to match analog-to-digital converters (ADC's) and DAC's and often must be adjusted according to calibration curves. Groups of digital signals must be encoded as discrete variables. Obviously, the correctness of the entire system depends critically upon such translations. Translations must be computed in both directions: register contents are encoded into variables on input, and state variables are decoded back to register contents on output.

It is also necessary to detect errors (where clients invoke operations with invalid parameters) and faults (where operations fail despite valid invocation by the client). In most computing applications, it is expected that programs should detect and report errors, but it is left to the users to deal with faults. Control systems differ from other computing applications in part because they are expected to handle some faults.

Our system is large, but most of its size derives from repetition of similar elements. We hope to produce a manageable system by basing the implementation on tables that describe the system configuration. For example, one table, the "wiring list," tells which signals are connected to particular input/output device register addresses; another table, the "crate diagram," tells which kind of device is present at each address. It should be possible to accommodate most configuration changes simply by editing tables, rather than changing executable code. Changing the routing between signals and devices, or adding additional signals and devices similar to those already present, should only require changing tables. Accommodating new kinds of signals and devices should only require adding the minimal amount of code needed to deal with their specific characteristics.

The requirements are not difficult to understand, but a large control system provides many opportunities for confusion and error. We decided it would be useful to conduct a formal

[1] I could find only [1] and [3], and these are not recent.

development. The purpose of the effort reported here was to determine what the tables should contain and how the implementation should interpret their contents. The objective was to produce a specification that was sufficiently detailed so that its implementation in an imperative programming language would be a straightforward exercise, and it would be clear how to construct and fill in the tables to describe any particular control system configuration. We intended that the development should be sufficiently rigorous that we could use proofs to support claims we would like to make, for example, "All possible inputs will be handled properly," or "Every output will be valid."

We used the Z notation [8] for this work because it is a good match to our chosen table-driven strategy. Global functions and relations in Z correspond to our configuration tables. Z deals well with partial functions, providing a natural model for tables with gaps where address spaces are not populated. Functional composition in Z can represent computations where an entry found in one table is used as an index into another. Many Z textbooks and case studies provide nice examples of error handling [6, 8]. Some of our preliminary experiments with Z appear in [4]. The Z text in this report was found free of syntax and type errors by the FUZZ type-checker [9].

Z does not provide built-in facilities for representing the passage of time, or synchronization of concurrent processes. This report deals only with the functional aspects of input/output, and defers other issues for treatment in a notation better suited for them.

This report is not a tutorial on Z, but the development is fully explained in the English text and is intended to be meaningful to readers who are not familiar with Z. In fact, this essay is intended to pursuade readers who are unfamiliar with formal methods that they can be practical and effective in real projects.

2 The initial specification

In this section we present the formal specification that serves as the starting point for our development.

2.1 The system configuration and the system state

At the lowest level, input/output is accomplished by data converter hardware. Data converters accomodate *signals* connected to the controlled process. Signals may be digital or analog, and carry information which is input or output with respect to the computer.

Data converters contain *registers*. Every register is identified by a unique *address*, and holds *contents* that encode the value of one signal in a form intelligible to the computer. In any particular configuration, a fixed set of addresses is populated with registers.

We distinguish register contents from control program *state variables*. While registers hold contents, *variables* have *values*.

Expressed formally, we have:

$$[ADDR, CONTENTS, VAR, VALUE]$$

$$\begin{array}{|l}
ioreg : \mathbb{P}\, ADDR \\
iovar : \mathbb{P}\, VAR \\
map : VAR \leftrightarrow ADDR \\
\hline
map(\!|iovar|\!) \subseteq ioreg
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Sys\;\hspace{20em}}} \\
register : ADDR \rightarrow\!\!\!\!\!\!\!+\;\; CONTENTS \\
variable : VAR \rightarrow\!\!\!\!\!\!\!+\;\; VALUE \\
\hline
\mathrm{dom}\, register = ioreg \\
\mathrm{dom}\, variable \cap \mathrm{dom}\, map \subseteq iovar \\
\hline
\end{array}$$

The sets *ioreg* and *iovar*, and the relation *map* model some of the tables that describe the configuration. The set *ioreg* lists the populated register addresses, *iovar* lists the variables that can be named as parameters in input or output operations, and *map* relates those variables to the registers from which they are derived. The predicate says that those variables must be related to registers that are populated.

Here *ioreg*, *iovar* and *map* are global constants that model the features of the system that do not change. We do not model any operations that change the configuration tables. We have actually specified a whole family of control systems, where each assignment of values to *ioreg*, *iovar* and *map* determines a particular control system that belongs to the family. This is an example of what Spivey [8] calls a *loose specification*.

The schema *Sys* models the volatile part of the system that can change state. The registers along with their contents are modeled as a function from addresses to contents, and the variables with their values are modeled as another function. The first predicate says that there is a fixed set of registers in the configuration; only their contents may change. There is no predicate fixing the domain of *variable*, because the set of variables may change as the program executes, as different subroutines are invoked or dynamic data structures evolve. It is possible that not all variables can be parameters of input/output operations; some may only store intermediate values in computations. However, any variable that may be input or output must be named in *iovar*.

Our choice of the words "register" and "address" does not mean this model is limited to memory-mapped devices. A register could be any distinguishable item of data, including a named element accessed through a "smart" controller, and an address might be a name derived from an English word, instead of a number. Moreover, values need not be scalar; variables might have complex internal structure.

2.2 Translating between register contents and state variable values

The higher-order function *encode* determines the translation between variable values and register contents. When applied to the *register*, *encode* returns a function from variables to values; changing the contents of the registers causes a different function to be returned. The operation schema *Translate* uses *encode* to calculate variable values from register contents (or vice versa).

$$encode : (ADDR \rightarrow\!\!\!\!\!\mapsto CONTENTS) \rightarrow\!\!\!\!\!\mapsto VAR \rightarrow\!\!\!\!\!\mapsto VALUE$$

Translate

$$\begin{array}{l} \Delta Sys \\ vars? : \mathbb{P}\ VAR \\ \hline variable' = variable \oplus vars? \lhd encode\ register' \end{array}$$

This schema can be specialized in each direction.

$$Encode \mathrel{\widehat=} [\ Translate \mid register' = register\ ]$$

$$Decode \mathrel{\widehat=} [\ Translate \mid variable' = variable\ ]$$

These operations are called *Encode* and *Decode*, not *Input* and *Output*, because they are only intended to model the translation between register contents and state variable values; they do not necessarily include the transfer of data between the registers and the transducers connected to the controlled process. In some systems, those transfers are separate operations.

We have not yet shown how the function *encode* is constructed. That is the subject of the following sections.

3 Development strategy

In this and following sections we show how the function *encode* is constructed from configuration tables, calibration formulas and other available information. A series of representations for *encode* are presented, where each successive version provides more detail that suggests a more straightforward or efficient implementation in an imperative programming language. This development process is often called *refinement*. An important advantage of a formal specification is that each development step can be calculated, inferred or otherwise formally justified.

Our development strategy is based on several observations:

- Usually only one, or a few, registers are related to any variable.
- The size of large systems often derives from repetition of similar elements.
- Each register and variable should only assume certain values.
- Functions can sometimes be replaced by sequences.
- Items defined non-constructively must be constructed later.

In the following sections we show how these observations are used, and introduce some functions and other items that we will use to construct *encode*.

3.1 Only consider elements that are related

The signature of *encode* suggests that all registers must be examined to determine the value of any variable. In fact, usually only one, or a few, registers are related to any variable. To use this observation, we must introduce more information that describes how the system is wired together.

Variables are ultimately derived from *signals* obtained from transducers that are connected to the controlled process. Each signal is connected to a particular register. The signal routing is recorded in voluminous wiring lists that are part of the documentation for any configuration. The relation *map* introduced in section 2.1 is composed from this information.

$[SIGNAL]$

$$
\begin{array}{|l}
signal : VAR \leftrightarrow SIGNAL \\
routing : SIGNAL \rightarrowtail\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!+ \;\; ADDR \\
\hline
map = signal \mathbin{⨾} routing
\end{array}
$$

This representation makes it easy to change *routing* as needed, for example to bypass a faulty data converter channel.

3.2 Collect similar elements into groups

Most of our system's size derives from repetition of similar elements; we can achieve efficiencies by grouping similar elements together and treating them all in the same way. This section introduces relations and functions that classify signals and variables into groups.

Each variable belongs to a *class*, and every instance of each class is composed of items called *members*. Each signal is associated with a particular member. In our system there are hundreds of variables and signals, but only a few dozen classes and members. Signals are assigned to variables such that all variables of the same class are associated with signals that have the same membership.

$$[CLASS, MEMBER]$$

$$
\begin{array}{|l}
class : VAR \mathrel{+\!\!\!\!\rightarrow} CLASS \\
member : SIGNAL \mathrel{+\!\!\!\!\rightarrow} MEMBER \\
classdef : CLASS \leftrightarrow MEMBER \\
\hline
\forall\, v : \mathrm{dom}\, class \bullet \\
\qquad classdef (\!| \{ class\ v \} |\!) = member (\!| \{ signal\ v \} |\!)
\end{array}
$$

Our choice of names is influenced by the "object-oriented programming" school; in fact we anticipate specifying another view of the system where the variables are instances ("objects") of abstract data types ("classes") defined by Z schemas.

It turns out to be useful to define the function *amember*, that associates each address with the membership of its attached signal, and *rmember*, that expresses registers in terms of membership rather than addresses:

$$
\begin{array}{|l}
amember : ADDR \mathrel{+\!\!\!\!\rightarrow} MEMBER \\
rmember : (ADDR \mathrel{+\!\!\!\!\rightarrow} CONTENTS) \mathrel{+\!\!\!\!\rightarrow} (MEMBER \mathrel{+\!\!\!\!\rightarrow} CONTENTS) \\
\hline
routing \mathbin{⨾} amember = member \\
\forall\, reg : \mathrm{dom}\, rmember \bullet \\
\qquad rmember\ reg = \{\, a : \mathrm{dom}\, reg \bullet amember\ a \mapsto reg\ a \,\}
\end{array}
$$

3.3 Limit the values that elements may assume

Each kind of signal and variable is only supposed to take on certain values. For analog quantities, these usually form a continuous range that represents physically reasonable values. For digital elements, these form a meaningful set of discrete indications.

$$
\begin{array}{|l}
mrange : MEMBER \leftrightarrow CONTENTS \\
crange : CLASS \leftrightarrow VALUE
\end{array}
$$

Register contents and variable values that do not lie within the ranges specified by these two relations are considered faulty.

The relation *mrange* constrains the contents of each register, considered by itself. Registers are sometimes collected together in groups, where the entire group encodes some variable. The schema *Combination* associates each class with all of its members and all combinations of their permitted values.

Combination

$$
\begin{array}{l}
cclass : CLASS \\
contents : MEMBER \rightarrow\!\!\!\!\!\!+\; CONTENTS \\
\hline
\operatorname{dom} contents = classdef \,(\!|\, \{ cclass \} \,|\!) \\
contents \subseteq mrange
\end{array}
$$

3.4 Replace functions with sequences

It is sometimes more efficient to replace functions with sequences. For many input/output devices it is usual to transfer the contents of a long sequence of consecutive addresses in a single operation (for example, in *direct memory access* (DMA) transfers). In such cases, we do not need to associate addresses with items, because the source of each item is indicated by its position in the sequence. Also, in most programming languages, parameters to operations (procedure calls, function applications etc.) are distinguished by their position in a sequence.

Therefore, for each class, we choose a particular sequence of the associated members. This determines a sequence of addresses associated with each variable.

$$
\begin{array}{|l}
member\_seq : CLASS \rightarrow\!\!\!\!\!\!+\; \operatorname{iseq} MEMBER \\
addr\_seq : VAR \rightarrow\!\!\!\!\!\!+\; \operatorname{iseq} ADDR \\
\hline
\forall\, t : \operatorname{dom} classdef \bullet \\
\qquad \operatorname{ran}(member\_seq\ t) = classdef (\!| \{ t \} |\!) \\
\forall\, v : \operatorname{dom} class \bullet \\
\qquad (addr\_seq\ v) \mathbin{;} amember = (class \mathbin{;} member\_seq)\ v
\end{array}
$$

3.5 Use non-constructive definitions

In Z, the specification for some item (such as a function) is constructive if the item appears by itself on the left side of an equal sign in the predicate where it is defined. Z also permits non-constructive specifications. This is an advantage because they can be concise and expressive. For example, the function *addr_seq* is defined non-constructively in the preceding section (3.4). However, at some point it is necessary to show how items so defined can be implemented.

4 The development steps

We now have enough pieces in place to perform the development.

4.1 Define the translation tables

We begin by defining the higher-order function *translate*, which provides a way for system designers to describe the translation function for each class. When *translate* is applied to some class c, it returns a function that associates values with certain patterns of members and contents.

$$
\begin{array}{|l}
translate : CLASS \rightarrow\!\!\!\!\!\!+\; (MEMBER \rightarrow\!\!\!\!\!\!+\; CONTENTS) \rightarrow\!\!\!\!\!\!+\; VALUE \\
\hline
\forall\, c : \operatorname{dom} translate \bullet \\
\qquad \operatorname{dom}(translate\ c) \subseteq \{\, Combination \mid cclass = c \bullet contents \,\} \wedge \\
\qquad \operatorname{ran}(translate\ c) \subseteq crange (\!| \{ c \} |\!)
\end{array}
$$

Using this predicate as a template, for any *translate c* we can use *Combination* to construct the domain of the function, and then use our knowledge of the application to assign each corresponding $VALUE$. For analog quantities it is usually effective to represent this assignment by a formula; for digital elements, a table is often preferred.

The predicate ensures that valid register contents are mapped into valid variable values (and vice-versa). Consequently, analog quantities will not overflow, saturate or clip when they are transfered in either direction. This is particularly important for output, because converters often provide quite limited resolution, but it is important to use as much resolution as is available. Designers must be especially mindful of this consideration when substituting converters during configuration changes.

Filling in *translate c* for each class is a central task in creating the detailed specification for a particular control system. This report only describes generic aspects of the specification, so this level of detail will not be described further. Our results are valid for any choice of the *translate c* that satisfy the predicate given here. That is what we mean when we say our system is table-driven.

The predicate does not require all possible patterns allowed by *Combination* to appear in the domain of each *translate c*. It is usual for some patterns to be omitted because they indicate faulty signal values or combinations. For example, it is good practice for binary conditions to be indicated by two independent signals, rather than a single binary signal. Thus, there would be independent signals to indicate *valve_open* and *valve_closed*, where each signal might indicate *set* or *clear*. The pattern

$$\{valve\_open \mapsto set, valve\_closed \mapsto clear\}$$

is expected and would be included in the domain of *translate valve*, but the pattern

$$\{valve\_open \mapsto set, valve\_closed \mapsto set\}$$

indicates at least one sensor is faulty and would be omitted. (Alternatively, a special element of $VALUE$ could be provided to indicate this and similar faults. This alternative would be handled differently by the fault-detection mechanism described in section 6.)

By first limiting the permitted ranges in *mrange* and *crange*, and then choosing which patterns of those defined by *Combination* belong to the domain of each *translate c* when we configure a system, we can determine which patterns of register contents represent valid encodings of any given set of variables. This is so important that we define a schema to describe it:

$$\begin{array}{|l}
\textit{RegValid} \\
\hline
vars? : \mathbb{P}\ VAR \\
reg : ADDR \rightarrow\!\!\!\!\!\!+\ CONTENTS \\
\hline
\mathrm{dom}\, reg = map(\!|vars?|\!) \\
\forall\, v : vars? \bullet \forall\, rs : \mathbb{P}\ reg \mid \mathrm{dom}\, rs = map(\!|\{v\}|\!) \bullet \\
\quad\quad rmember\ rs \in \mathrm{dom}\,(translate\ (class\ v)) \\
\hline
\end{array}$$

This schema is used in subsequent developments to describe correctness (section 5) and specify faults (section 6).

4.2 Define *encode*

Next, we propose a definition for *encode*. If we already know the class c that a variable belongs to, and which registers reg determine the variable's value, *encode* can be derived from *translate* by using *rmember* to adjust the type of *reg*:

$$encode\ reg\ v = translate\ c\ (rmember\ reg)$$

In fact v is related to c and reg through functions *class* and *map*, respectively, so:

$$encode\ register\ v = translate\ (class\ v)\ (rmember\ (map(\!|\{v\}|\!) \lhd register))$$

4.3 Make *encode* more efficient

In this development step, we increase efficiency by replacing functions with sequences. From the function *translate* we derive *translate_seq*:

$$\begin{array}{|l}
translate_seq : CLASS \rightarrow\!\!\!\!\!\!+\ \mathrm{iseq}\ CONTENTS \rightarrow\!\!\!\!\!\!+\ VALUE \\
\hline
\forall\, c : \mathrm{dom}\, translate \bullet translate_seq\ c = \\
\quad\quad \{\, ccontents : \mathrm{dom}\,(translate\ c) \bullet \\
\quad\quad\quad\quad member_seq\ c \mathbin{;} ccontents \mapsto translate\ c\ ccontents \,\}
\end{array}$$

Then *encode* can be expressed:

$$encode\ register\ v = translate\_seq\ (class\ v)\ (addr\_seq\ v \mathbin{;} register)$$

This suggests an efficient implementation in an imperative programming language (as described in section 8).

4.4 Make definitions constructive

In this final development step we derive constructive specifications from non-constructive ones.

The expression $code\_seq\,(class\ v)\,(addr\_seq\ v \fatsemi register)$ derived in the previous step almost resembles an executable statement in a functional programming language[2]. However, this expression could not be evaluated because we have not yet shown how to construct the function $addr\_seq$. It is defined non-constructively in section 3.4:

$$(addr\_seq\ v) \fatsemi amember = (class \fatsemi member\_seq)\ v$$

First, we must deal with $amember$, which is also defined non-constructively (section 3.4):

$$routing \fatsemi amember = member$$

Here $amember$ is defined to be the function which, when composed with $routing$, yields the function $member$. This definition is convenient because it makes sense for designers to choose $member$ and $routing$; $amember$ is determined by those choices. However, it does not show how to construct $amember$. For that, we use a lemma about functional composition and inverse:

$$f \fatsemi g = h \Rightarrow g = f^{\sim} \fatsemi h$$

This lemma is proved in Appendix A. We obtain:

$$amember = routing^{\sim} \fatsemi member$$

Since $routing$ is an injective function, its inverse is also a function, which ensures $amember$ is functional. The definition of $addr\_seq$ expands to:

$$(addr\_seq\ v) \fatsemi routing^{\sim} \fatsemi member = (class \fatsemi member\_seq)\ v$$

We would like to transform this to a constructive definition by twice applying another lemma:

$$f \fatsemi g = h \Rightarrow f = h \fatsemi g^{\sim}$$

[2] Perhaps such resemblances help explain the common misconception that a formal specification is just a program written in a very high-level language.

However, $member^{\sim}$ is not functional because *member* is not injective; many signals have the same membership. If we applied $member^{\sim}$ to the expression on the right, we would obtain a relation associating multiple signals where we just want one. Fortunately, it follows from the definitions of *class* and *classdef* (section 3.2) that the signals that contribute to a single variable do have different membership. Thus we can use domain restriction to obtain a constructive definition of *addr_seq*:

$$addr\_seq\ v = (class \mathbin{⨾} member\_seq)\ v \mathbin{⨾} (signal(\!|\{v\}|\!) \lhd member)^{\sim} \mathbin{⨾} routing$$

This completes the development.

Non-constructive definitions are one of the essential features that distinguish specification notations from programming languages. If we were limited to a constructive notation, a definition for *addr_seq* would have to be assumed without justification. It probably would have been divined by intuition, outside the formal development process. Sometimes intuition works, sometimes not. What if *routing* were not injective? What if we did not notice that $member^{\sim}$ is not functional?

5 Checking the development by calculation and proof

An important advantage of formal specifications is that they can be analyzed by performing calculations or sound logical inferences. From the specification, it should be possible to prove conjectures that formally express properties or qualities that we wish the system to have. This can provide confidence that the formal specification really does express the intended requirements.

Another use of proof is to check developments. We are fallible, so any development step might be wrong[3]. If the development is correct, it should be possible to prove that the product of each step is consistent with all of its predecessors. This ability to check each intermediate development step by calculation and proof, rather than having to wait until an executable program can be produced and tested, distinguishes formal development from more intuitive software development methods.

We find that the conjecture we pose to express some informal requirements also serves to check our entire development sequence. We would like to claim,

"All possible inputs will be handled properly"

[3]In fact, some of the development steps are based on intuition (i.e. guessing) and should be checked.

but this statement is much too vague to relate to our formal specification. Trying again, we say,

> "If the configuration tables are accurate, then valid input signals will be translated to the proper state variable values, and invalid input signals will be detected."

Here the features of the formal specification begin to emerge, but we need still more detail. Trying once again, we begin,

> "If a group of signals are entered into the routing table, and the addresses to which the signals are connected are populated, and the signals are found in the tables to be members of recognized classes, and the value of each signal falls within the permitted range for its class membership, and . . . "

This is beginning to assume the prolix but shallow quality of most theorems in software verification. Rather than press on in this vein, we resort to formal notation. We are trying to state a hypotheses about signals:

$$
\begin{array}{|l}
\hline
\textit{SigValid} \\
\hline
sigs : \mathbb{P}\, SIGNAL \\
scontents : SIGNAL \mathrel{+\!\!\!\!\to} CONTENTS \\
\hline
sigs = \operatorname{dom} scontents \\
routing(\!| sigs |\!) \subseteq ioreg \\
member(\!| sigs |\!) \subseteq \operatorname{ran} classdef \\
\exists\, RegValid \bullet \\
\qquad sigs = signal(\!| vars? |\!) \land \\
\qquad scontents = \{\, a : \operatorname{dom} reg \bullet routing^{\sim} a \mapsto reg\ a \,\} \\
\hline
\end{array}
$$

Here, we use the *RegValid* schema defined in section 4.1. We wish to relate the signals *sigs* in these hypotheses to the *vars*? parameter in the *Encode* operation schema, so we also need:

$$
\begin{array}{|l}
\hline
\textit{SigMatchVar} \\
\hline
sigs : \mathbb{P}\, SIGNAL \\
vars? : \mathbb{P}\, VAR \\
\hline
sigs = signal(\!| vars? |\!) \\
\hline
\end{array}
$$

Now we can state the entire conjecture, which formalizes the requirement, "All valid signals will be handled properly".

$$
\begin{array}{l}
\mathit{SigValid} \land \mathit{SigMatchVar} \land \mathit{Encode} \vdash \\
\quad \forall\, v : \mathit{vars}? \bullet \exists\, c : \mathit{CLASS} \bullet c = \mathit{class}\ v \land \\
\qquad \mathit{variable}'\ v = \mathit{translate}\ c\, \{ s : \mathit{sigs} \bullet \mathit{member}\ s \mapsto \mathit{scontents}\ s \}
\end{array}
$$

The conclusion says that the variables should have the values required for the signals' contents by the tables and formulas for the signals' class membership.

This example demonstrates Z style for writing conjectures [6]. It means that, from the declarations and predicates of the schema expression before the turnstile $\vdash$, we can derive (or prove) the conclusion after the turnstile[4].

Proving this conjecture would provide confidence that the formal specification expresses the informally stated requirement[5]. It would also serve as a good check on the development steps, since the hypothesis *SigValid* and the conclusion are expressed in terms of signals, classes, members, and the function *translate*, while the schema that describes the operation, *Encode*, is expressed in terms of registers, variables, and the function *encode*. The schema *SigMatchVar* ties the two representations together.

Our development, and the conjecture posed to check it, are closely related to the formal development method called *refinement* described by Spivey [8] and taught (in slightly different form) by Potter, et al. [6]. In this method, one first proposes an *abstract* specification, then *refines* it to a more *concrete* specification composed of items that are closer to the implementation language. The *retrieve relation* relates items in the abstract and concrete states. To justify a refinement, one should prove several conjectures that the method prescribes. One of these, the *correctness conjecture*, has the form:

$$
\mathrm{pre}\ \mathit{AOp} \land \mathit{Retr} \land \mathit{COp} \land \mathit{Retr}' \vdash \mathit{AOp}
$$

Where *AOp* is the operation schema for the abstract state, *Retr* is the schema describing the retrieve relation, and *Cop* is the operation schema for the concrete operation[6].

This is almost the same as our own conjecture; our hypotheses *SigValid* corresponds to pre *AOp*, our *SigMatchVar* to *Retr*, our *Encode* to *COp*, and our conclusion to *AOp*. The similarity is clear, except our development proceeded from the "concrete" to the "abstract"

[4]This convention is not described in [8], and this conjecture was not type-checked.
[5]I have not yet attempted to prove any of the conjectures posed in this report.
[6]This is the form used in [6]. They call this the *correctness theorem*, not *conjecture*.

represention! We shouldn't feel obligated to follow the recommended sequence too dogmatically, but instead should consider refinement a useful collection of methods for checking development steps, regardless of the order in which they are discovered.

6 Detecting errors and faults

An important specification task is to enumerate the errors and faults that might occur in each operation and determine how to handle them. An advantage of a formal specification is that it isn't necessary to rely solely on inspiration to discover potential errors and faults; some of them can be calculated.

6.1 Calculating the preconditions

The *precondition* of an operation describes the set of initial states for which a final state is defined. If the precondition of an operation does not include some possible states, the operation is *partial.* States which do not satisfy the precondition of a partial operation are usually those that designers consider erroneous or faulty[7]. Obviously, it is poor practice to implement partial operations, since their response to errors and faults is undefined.

Our *Encode* operation is partial, as the following calculation reveals. In Z, it is not necessary to explicitly state preconditions. Preconditions that are implicit in an operation schema can be calculated, as described in [8]. For the operation *Encode* on state *Sys*, the precondition is:

$$PreEncode \mathrel{\widehat{=}} \exists\, Sys' \bullet Encode$$

The essential requirement expressed here turns out to be that the expression $vars? \lhd encode\ register$ from *Encode* must be defined, so *register* must lie within the domain of *encode*, and *vars*? must lie within the domain of $encode\ register$. The expression on the right can be expanded, simplified and expressed as a schema[8]:

[7]If a state fails to satisfy a precondition, but does not appear erroneous or faulty, the formal specification may not express the intended requirements.

[8]This is a guess. I haven't actually performed the calculation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PreEncode \quad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Sys \\
vars? : \mathbb{P}\ VAR \\
\hline
vars? \subseteq iovar \\
\exists\, reg : \mathbb{P}\ register \bullet RegValid \\
\hline
\end{array}$$

The first line in the predicate deals with the input parameter *vars*?; failure to satisfy this predicate is an error. The second deals with the contents of *register*; failure to satisfy this predicate is a fault. The latter predicate is based on the schema *RegValid* defined in section 4.1.

6.2 Making the operation total

With preconditions in hand, we can follow usual Z style for extending partial operations. Errors and faults are reported through *status* values. We define a small schema to indicate an operation was successful.

$$STATUS ::= success \mid badvar \mid badreg \mid \ldots$$
$$Success \mathrel{\widehat{=}} [\, status : STATUS \mid status = success \,]$$

When an error or fault is detected, it should be reported, but the system state must not otherwise change, for example:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BadReg \quad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Sys \\
vars? : \mathbb{P}\ VAR \\
status : STATUS \\
\hline
status = badreg \\
\neg\, (\exists\, reg : \mathbb{P}\ register \bullet RegValid) \\
\hline
\end{array}$$

A similar schema *BadVar* describes the case where the input variables are not valid. Then the *Encode* operation can be extended:

$$T\_Encode \mathrel{\widehat{=}} (Encode \land Success) \lor BadVar \lor BadReg$$

These three outcomes are supposed to exhaust all possibilities. Therefore, the operation *T_Encode* should be *total*; it should be defined in every possible state. This could be confirmed by proving $Pre\_T\_Encode \Rightarrow Sys$.

7 Inverse operations

Non-constructive notations make it easy to define inverse operations. One example is the definition of the *Decode* operation that translates state variable values to register contents (section 2.2). It contains the predicate

$$variable = variable \oplus vars? \lhd encode\ register'$$

This is nonconstructive because $register'$ is the item being defined. We can define a constructive version of *Decode* by analogy with *Encode*:

$$decode : (VAR \pfun VALUE) \pfun ADDR \pfun CONTENTS$$

Decode

$$\Delta Sys$$
$$vars? : \mathbb{P}\ VAR$$

$$variable' = variable$$
$$register' = register \oplus map(\!|vars?|\!) \lhd decode\ variable$$

It seems clear that *decode* should be the inverse of *encode*. If we already knew which variable v was associated with a particular register address, we could derive *decode* from the inverse of *translate*:

$$decode\ variable\ a = (translate\ (class\ v))^{\sim}\ (variable\ v)\ (amember\ a)$$

In fact v is related to a through the function *map*, so we can just use the preceding definition, where it is understood that $v \in map^{\sim}(\!|\{a\}|\!)$ (any element will do)[9].

The function *decode* is expressed here in terms of previously-defined items. Thefore, we do not need to define any additional configuration tables. Where evaluating *translate c* is implemented by table lookup, the implementation of $(translate\ (class\ v))^{\sim}$ can use the same table, with the direction of lookup reversed.

This development could be checked by proving $Encode \fatsemi Decode \vdash register' = register$.

[9] Hmnn. . .

8 Towards an implementation

In this section show how the final development of our specification suggests an efficient and straightforward implementation in an imperative programming language.

Given sets, global functions and relations can be implemented as tables that are loaded when the control system is initialized; the predicates for these items suggest acceptance tests that could be performed when the tables are produced, or run-time checks to be performed each time the tables are loaded. Application of a higher-order function to its first argument can be implemented as code that dispatches to a handler for that particular case. Ordinary function application can be implemented as table lookup or formula evaluation, whichever is more efficient.

In section 4.3 we showed that the function *encode* can be represented:

$$encode\ register\ v = translate\_seq\ (class\ v)\ (addr\_seq\ v \fatsemi register)$$

Evalution of *encode register v* could be implemented as follows: Look up the sequence of register addresses associated with variable v, which is $addr\_seq\ v$. Then find the contents of those registers, preserving the sequence order; that is $addr\_seq\ v \fatsemi register$. Look up the class of v, which is $class\ v$. Then dispatch to the code that handles that class, which is $translate\_seq\ (class\ v)$; this code usually looks up items in tables or computes formulas. Finally, execute that code on the sequence of register contents. That is expressed by $translate\_seq\ (class\ v)\ (addr\_seq\ v \fatsemi register)$, and yields $variable\ v$, the value of v.

Functional composition (indicated $f \fatsemi g$) is ubiquitous in this specification. In some cases, for example *addr_seq* as defined in section 4.4, it is clearly more efficient to pre-compute the composed function when a system is configured. In other cases, the specification reveals opportunities to trade off time against space. For example, the expression $translate\_seq\ (class\ v)$ could be implemented in two steps, as described previously. The equivalent representation $(class \fatsemi translate\_seq)\ v$ suggests the alternative of speeding execution by first pre-computing a single large table with an entry for each variable (not just each class).

The final development of the specification appears sufficiently detailed to serve as a basis for formal verification of the implementation, if we wish to attempt it.

9 Discussion

Critics charge that many formal developments are described in retrospect, merely casting into formal notations work that has already been developed intuitively. Our own experience refutes that cynical assessment.

Our results may seem obvious; in fact, we made several false starts that led to cumbersome specifications and unfinished developments. The formal notation, by providing compact descriptions that can be manipulated and checked algebraically, was quite helpful for identifying problems, exploring alternatives, and expressing a detailed solution. The development was easy to calculate, but might have been difficult to intuit or improvise. Had we plunged on into implementation without the formal development, we might have begun building a less satisfactory alternative.

The existing implementation provides some indication of the difficulties we have avoided (we are developing a replacement for the control system that was provided by the cyclotron vendor when the facility was installed [7]). The code seems unnecessarily large and is quite difficult to follow. There was some attempt to make the system table-driven, but reconfiguring it to accommodate different converter hardware, or even to move a signal from a failed converter, cannot be accomplished by changing table entries; it is necessary to modify executable code. There is little error checking; it is not clear what errors are checked. A manual startup sequence is necessary because the sytem can set outputs before valid register contents are established.

The formally-developed replacement described in this report has not yet been implemented, but we are confident that it will not be difficult, and will provide much improvement over the present system.

References

[1] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proceedings, Fifth International Conference on Software Engineering*, pages 195–204. IEEE Computer Society Press, 1981.

[2] Dan Craigen. FM89: Assessment of formal methods for trustworthy computer systems. In *12th International Conference on Software Engineering Proceedings*, pages 233–235. IEEE Computer Society, 1990.

[3] K.L. Heninger. Specifying software requirements for complex systems: new techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, 1980.

[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, 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.

[6] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall International (UK) Ltd, Hemel Hempstead, Hertfordshire, 1991.

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

[8] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, New York, 1989.

[9] J. M. Spivey. *The FUZZ Manual*. J. M. Spivey Computing Science Consultancy, Oxford, January 1991. Second Printing.

A Two useful lemmas about composition and inverse

In this appendix we prove two lemmas used in section 4.4 to derive constructive definitions from non-constructive ones.

Our proofs are based on the observation that composition with some function is itself a function (a higher order function that takes a function as input and returns a new function). This is true regardless of the order in which the functions are composed; either of the two composed functions can be considered the argument of a higher-order function. Therefore, we can apply Leibniz' law, which states that

$$a = b \Rightarrow f\ a = f\ b$$

for any objects a and b, where f is any function with the appropriate signature.

First we prove $f \mathbin{;} g = h \Rightarrow g = f^{\sim} \mathbin{;} h$. Page numbers refer to [8].

$f \mathbin{;} g = h$ [premise]

$f^{\sim} ⨾ (f ⨾ g) = f^{\sim} ⨾ h$ [Leibniz]

$(f^{\sim} ⨾ f) ⨾ g = f^{\sim} ⨾ h$ [law about composition, p. 97]

$\mathrm{id}\,(\mathrm{ran}\, f) ⨾ g = f^{\sim} ⨾ h$ [law about id, ran, inverse and composition, p. 105]

$g = f^{\sim} ⨾ h$ [law about id and composition p. 97]

The proof of $f ⨾ g = h \Rightarrow g = h ⨾ g^{\sim}$ is similar:

$f ⨾ g = h$ [premise]

$(f ⨾ g) ⨾ g^{\sim} = h ⨾ g^{\sim}$ [Leibniz]

$f = h ⨾ g^{\sim}$ [laws about composition, inverse, etc. pps. 97 and 105]