# FORMAL SPECIFICATIONS FOR A CYCLOTRON CONTROL SYSTEM

Jonathan Jacky and Ruedi Risler

Department of Radiation Oncology,
University of Washington,
Seattle, WA 98195, USA

## ABSTRACT

We are preparing a comprehensive specification for the computer control system of a cyclotron and treatment facility that provides particle beams for cancer treatments with fast neutrons, production of medical isotopes, and physics experiments. The informal (prose) specification describes the control system as thoroughly as is practical using standard technical English, supplemented by tables, diagrams, and some algebraic equations. We are supplementing the prose specification with a formal (mathematical) specification. Potential advantages of formal specifications are discussed, and experiences writing and using formal specifications are described.

## 1. INTRODUCTION

The Clinical Neutron Therapy System (CNTS) 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. The facility was installed in 1984, and includes a computer control system provided by the cyclotron vendor.[1] The University is now developing a new, successor control system. This development project is motivated by requirements to make the system easier and quicker to use, easier to maintain, and able to accomodate future hardware and software modifications.[2]

We are attempting to achieve high reliability and safety by applying rigorous software development and quality assurance practices. Modern recommendations for the development of safety-critical computer-based systems[3,4] emphasize the need for an explicit statement of functional and safety requirements, and a development process that can be shown to produce an implementation that meets the requirements. A central idea is that developers should provide a functional specification that allows system outputs to be predicted if system inputs are described.

We determined that our first step in this project should be the production of a comprehensive specification for the new control system, to serve as an authoritative and complete guide for software development, testing, and instruction of facility users.

## 2. THE INFORMAL SPECIFICATION

We first prepared an informal (prose) specification, expressed in standard technical English, supplemented by tables, diagrams, and some algebraic equations.

The informal specification consists of four parts. Part I is an overview of the system that describes the facility, the hardware organization of controls, and introduces much of the vocabulary used in subsequent parts. It comprises almost 100 pages of single-spaced text. Part II is a detailed specification of operations which users perform at video terminals and control consoles. It comprises over 250 pages, including many illustrations of displays. Part III will be a detailed specification of internal operations involving the cyclotron and therapy apparatus itself, which are only indirectly visible to users, and will comprise about 100 pages. Part IV will be a largely tabular presentation of numerical operating parameters whose values can be changed independently of the rest of the specification. We plan to maintain these tables in a relational database.

Parts I and II have been prepared for distribution outside our department as technical reports[5,6] and we anticipate releasing the other parts as they are completed in coming months.

## 3. FORMAL SPECIFICATIONS

The prose specification is organized to help facility users read and understand it, so they can offer meaningful reviews. However, this organization is not necessarily the best for purposes of programming and analysis. An alternative is to make greater use of mathematical nota-

tions, annotated sparingly with English. Such specifications are called *formal* specifications.

Formal specifications are distantly related to popular design notations based on data flow diagrams, which have been applied to acclerators.[7] We are investigating notations that provide more mathematical rigor, in the sense that they allow more properties to be inferred or calculated.

There are potential advantages to developing programs from formal specifications. They can be more precise and compact than English specifications, and can be automatically checked for certain kinds of errors. They can be used to prove or calculate whether the specificied behavior is consistent with certain intended properties, such as safety. Formal specifications can support systematic software development methods where every development step can be justified and checked.

## 4. A FORMAL CASE STUDY

We have prepared a formal specification for computations that calculate control system state variables from input/output device register contents (and vice-versa). Specified behaviors include scaling and offsetting analog quantities to match data converters (ADC's and DAC's), adjusting analog quantities according to calibration curves, encoding groups of digital signals as discrete variables, and detection of errors (where clients invoke operations with invalid parameters) and faults (where input/output devices report inconsistent data).

Our specification is motivated by our facility but is quite generic and should be widely applicable. It is parameterized so that an implementation can be adapted to different control systems by providing tables of configuration data, rather than changing executable code. The specification is not merely descriptive, but is also used in the formal development of a detailed design.

We used a formal notation called Z[8,9] (pronounced *zed*) for this work because it is a good match to our chosen table-driven strategy. Some of our preliminary experiments with Z have appeared,[10] and we have prepared a detailed report of the present work.[11] Excerpts appear in the following subsections. This brief report does not permit much explanation, so these excerpts will not be meaningful to readers unfamiliar with Z; they are only included to show what a formal specification looks like.

### 4.1. 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 in Z, we have:

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

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

*Sys*

$$\begin{array}{|l} register : ADDR \pfun CONTENTS \\ variable : VAR \pfun VALUE \\ \hline \mathrm{dom}\ register = ioreg \\ \mathrm{dom}\ variable \cap \mathrm{dom}\ map \subseteq iovar \end{array}$$

The sets *ioreg* and *iovar*, and the relation *map* model some of the tables that describe the configuration. 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.

### 4.2. Translating between Registers and State Variables

The function *encode* determines the translation between variable values and register contents. The operation *Translate* uses *encode* to calculate variable values from register contents (or vice versa):

$$\begin{array}{|l} encode : \\ (ADDR \pfun CONTENTS) \pfun VAR \pfun VALUE \end{array}$$

*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.

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

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

### 4.3. Formal Development

We have not yet shown how the all-important function *encode* is constructed. In the full report[11] we construct the function *encode* from configuration tables, calibration formulas and other available information. A series of representations for *encode* are presented, where each successive version provides more detail. We find that *encode* can be expressed:

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

This suggests an efficient implementation in an imperative programming language, as explained in the full report.[11]

### 4.4. Non-constructive Definitions

Formal specifications bear a superficial resemblance to executable programs, but there is an important difference: programs must be constructive, but specifications can be *non-constructive*. This is an advantage because non-constructive definitions can be concise and expressive. For example, they make it easy to define inverse operations, as in the definition of the *Decode* operation that translates state variable values to register contents (section 4.2). However, at some point it is necessary to show how items so defined can be implemented; this requires converting them to constructive form, as demontrated in the full report.[11]

### 4.5. Handling Errors and Faults

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

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. The *precondition* of an operation describes the set of initial states for which a final state is defined. States which do not satisfy the precondition of an operation are usually those that designers consider erroneous or faulty. It is poor practice to implement operations that admit such states, since their response to errors and faults is undefined. For our operation *Encode*, the precondition is:

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

The expression on the right can be expanded:

*PreEncode*

$$Sys$$
$$vars? : \mathbb{P}\ VAR$$
---
$$vars? \subseteq iovar$$
$$\exists reg : \mathbb{P}\ register \bullet RegValid$$

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.

We must extend the specification to deal with such conditions. When an error or fault is detected, it should be reported, but the system state must not otherwise change, for example:

*BadReg*

$$\Xi Sys$$
$$vars? : \mathbb{P}\ VAR$$
$$status : STATUS$$
---
$$status = badreg$$
$$\neg\ (\exists reg : \mathbb{P}\ register \bullet RegValid)$$

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

$$T\_Encode \mathrel{\widehat=} (Encode \wedge Success) \vee BadVar \vee BadReg$$

These three outcomes exhaust all possibilities. Therefore, the operation *T_Encode* is defined in every possible state.

### 4.6. Checking the Development

An important advantage of formal development is that each step can be calculated, inferred or otherwise formally justified, and then checked. This ability to check each intermediate development step, rather than having to wait until an executable program can be produced and tested, distinguishes formal development from more intuitive software development methods.

In addition to checking each step in turn, the entire development can be checked against the original requirements. For example, we would like to claim,

> "All possible inputs will be handled properly"

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 ..."

Rather than press on in this vein, we resort to formal notation. We are trying to state a hypotheses about signals:

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

Other parts of the hypothesis can be formalized in the same way. A formal conjecture which expresses the requirement, "All valid signals will be handled properly", is:

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

Proving this conjecture could check for errors in the development, and provide confidence that the formal specification expresses the intended requirements.

## 5. FUTURE WORK

The first formally-specified subsystem scheduled for implementation is the controller for the leaf collimator that is used to shape the therapeutic neutron beam.[2] The program will be written in a Pascal dialect[12] and will run on a Digital Equipment Corporation MicroVAX II computer.

## 6. REFERENCES


1) 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, IEEE, East Lansing, Michigan, May 1984.
2) R. Risler, P. Wootton, F. Ziai, J. Jacky, S. Brossard, and I. Kalet. Continued operation of the Seattle clinical cyclotron facility. (This proceedings).
3) Great Britain Health and Safety Executive. *Programmable Electronic Systems in Safety Related Applications. Volume 1: An Introductory Guide. Volume 2: General Technical Guidelines.* Her Majesty's Stationery Office, London, 1987.
4) Nancy G. Leveson. Software safety: what, why and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.
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) 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.
7) G. A. Ludgate, B. Haley, L. Lee, and Y. N. Miles. The use of structured analysis and design in the engineering of the TRIUMF data acquisition and analysis system. *IEEE Transactions on Nuclear Science*, NS34(1):157 – 161, 1987.
8) J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, New York, 1989.
9) Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall International (UK) Ltd, Hemel Hempstead, Hertfordshire, 1991.
10) 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).
11) Jonathan Jacky. *Formal Specification and Development of Control System Input/Output.* Technical Report 92-05-02, Radiation Oncology Department, University of Washington, Seattle, WA, May 1992.
12) Digital Equipment Corporation. *VAXELN Technical Summary.* Digital Equipment Corporation, Maynard, MA, 1984.