STRUCTURING REQUIREMENTS
SPECIFICATIONS

by

Vittu Khanna

B.Sc. (Physics), Panjab University, India, 1991

M.Sc.(Physics), California Institute of Technology, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1998

©Vittu Khanna, 1998
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.
Abstract

Structure is used to enhance the understandability of source code comprising a software system. To study whether structure may enhance the understandability of software requirements specifications, this thesis investigates the application of two source code comprehension tools, Software Reflexion Models and Rigi, to requirements specifications. Two example specifications are studied using the tools. Further, a component (or scope) primitive for the Z specification language is proposed to help add structure to requirements. The example specifications in the extended Z have been rewritten using the component primitive. We conclude that this extension may not only help in increased comprehension but also in the prevention of some errors.
# TABLE OF CONTENTS

## Abstract

Table of Figures vi

Acknowledgments vi

1. INTRODUCTION 8

1.1 Motivation 8

1.2 Overall Method 4

1.3 The example specifications
   1.3.2 The Radiation Therapy Machine 6
   1.3.3 Why we used these particular specifications 6

1.4 The Tools 7

1.5 Thesis Outline 8

2. REQUIREMENT ANALYSIS: SPECIFICATION AND STRUCTURE 9

2.1 Introduction 9

2.2 Requirement Analysis and Specification
   2.2.1 Requirement Analysis 10
   2.2.2 Requirements Specification 11

2.3 Different Types of Specifications
   2.3.1 Ad-Hoc Natural Language Specification Approach 13
   2.3.2 Semi-Formal Methods 13
   2.3.2.1 Structured Analysis 13
   2.3.3.2 Threads-Based Technique 14
   2.3.3 Object-Oriented Methods 14
2.3.4 Formal Notation

2.4 Formal methods
   2.4.1 Formal Specification Language, Z

2.5 Structuring Requirements
   2.5.1 The Change Process
   2.5.2 Structure of a System
      2.5.2.1 Modularity
   2.5.3 Structuring Primitives
      2.5.3.1 Refinements
      2.5.3.2 Abstraction
      2.5.3.3 Summarization

2.6 Summary

3. STRUCTURE COMPREHENSION TOOLS
   3.1 Introduction
   3.2 RIGI
   3.3 Software Reflexion Models
   3.4 Using Source Code Comprehension Tools for Requirements
   3.5 Experimentation
   3.6 Summary

4. COMPREHENDING SOFTWARE REQUIREMENTS
   SPECIFICATIONS
   4.1 Introduction
   4.2 Rigi
      4.2.1 The Different Levels
   4.3 Software Reflexion Model
      4.1.1 Observations and their Implication
   4.3 Software Reflexion Model and Rigi: A comparison
   4.3 Summary

5. STRUCTURED SPECIFICATIONS
   5.1 Introduction
   5.2 The two structured specifications
      5.2.1 The Telereg specification
5.2.2 The Radiation Therapy Machine specification

5.2 Summary

6. RELATED WORK

7. CONCLUSION
   7.1 Future Work

BIBLIOGRAPHY

APPENDIX A Source model for telereg in the Software Reflexion Model

APPENDIX B The mapping file for the Software Reflexion Model

APPENDIX C The high level model for the Software Reflexion Model

APPENDIX D Telereg - structured specification

APPENDIX E Radiation therapy machine -
   Structured specification
Table of Figures

Figure 2.1: Uses of a Requirement Specification[7, p.14] .................................................. 12
Figure 3.1: Software Reflexion Model[25-p.20] .................................................................. 30
Figure 4.1: Level 1 .............................................................................................................. 39
Figure 4.2: the source model for the call relation .............................................................. 40
Figure 4.4: Level 2 .............................................................................................................. 41
Figure 4.5: Level 3 .............................................................................................................. 42
Figure 4.6: Level 4 .............................................................................................................. 43
Figure 4.7: Level 5 .............................................................................................................. 44
Figure 4.8: The entire ‘structured’ telereg specification .......................................................... 45
Figure 4.9: Unstructured Telereg specification .................................................................... 47
Figure 4.20: Level 1 ........................................................................................................... 48
Figure 4.11: Level 2 .......................................................................................................... 50
Figure 4.12: Level 3 .......................................................................................................... 51
Figure 4.13: Level 4 .......................................................................................................... 52
Figure 4.15: The Delta relation in the Telereg specification .................................................. 54
Figure 4.16: The call/include relation - Radiation Therapy Machine ................................. 55
Figure 4.17: The Xi relation - Radiation Therapy Machine .................................................. 56
Figure 4.18: The Delta relation - Radiation Therapy Machine ............................................. 57
Figure 4.19: Telereg specification viewed syntactically ....................................................... 58
Figure 4.20: The Include relation in the Telereg specification .............................................. 59
Figure 4.21: Reflexion Model of Telereg Specification .......................................................... 60
Figure 4.22: Reflexion Model of Radiation Therapy Machine Specification ....................... 61
Figure 4.23: Division of the Telereg .................................................................................... 63
Figure 4.24: A more detailed view ...................................................................................... 63
Figure 4.25: The most detailed specification of the Telereg system. As expected there are no
relations between telereg_state and global_types........................................................... 65
Figure 4.26: The Telereg System - Typed Reflexion Model .................................................. 67
Figure 4.27: Typed Reflexion Model - Radiation Therapy Machine ...................................... 69
Figure 4.28 .......................................................................................................................... 70
Acknowledgments

I would like to thank Dr. George Tsiknis for his help and guidance at the various stages of research and for his invaluable comments and criticisms on the thesis.

I would also like to thank Dr. Gail Murphy for carefully going through this document and for her helpful comments as well as useful answers to my numerous questions on the Software Reflexion Model tool.

In addition, I am grateful to Jonathan Jacky, Johannes Martin and Kenny Wong for their prompt replies to my email queries on the Radiation Therapy Machine and the Rigi tool.

Finally, I would like to express my gratitude to my family and friends, who made this possible for me. I would like to thank my parents for always believing in me and my brother who helped me tremendously. I would like to express my deep gratitude to my spouse, Yogesh, not only for his precious bits of advice, but also for the nights he spent in the lab, helping me in my thesis work. I could not have done it without his help.
Chapter 1

Introduction

"Where shall I begin, please your majesty?" he asked.
"Begin at the beginning", the king said gravely,
and go on till you come to the end: then stop."
-Lewis Carroll, Alice’s Adventures in Wonderland, Ch. 11

1.1 Motivation

More than 50% of software evolution work is devoted to system understanding[32]. As the overall system becomes large and complex, system structure becomes an issue[32]. For a large system, understanding the structural aspects of the system’s architecture is more important than understanding any single algorithmic component. Documentation that exists for these large systems usually describes isolated
parts but not the overall architecture. An overall view of the system structure helps in system understanding and, therefore, may ease the other software engineering phases. Software structure may be defined as follows:

"Software structure is the collection of artifacts used by software engineers when forming mental models of software systems. These artifacts include software components such as procedures, modules, interfaces and subsystems; dependencies among components such as client-supplier, inheritance and control-flow; and attributes such as component-type, interface size, and interconnection strength. A system's structure is the organization and interaction of these artifacts"[34, p.47].

In order to structure the system, some concepts used are subsystem structures; layered structures; aggregation, generalization, specialization and inheritance hierarchies; resource-flow graphs; component and dependency classification; event-handling strategies; and client-server and distributed architectures[2].

At the beginning of most software projects, the customer provides the contractor with a statement of requirements. Typically, the customer statement of requirements is expressed in natural language using technical terms of the customer application domain. The contractor will derive a requirement specification from the customer statement of requirements that describes the requirements in more detail. The process of identifying and understanding the customer requirements is called requirements analysis. This analysis phase also requires that the contractor determine the feasibility of the proposed system.

The Requirement Specification is the end product of the requirement analysis phase in the software development cycle. The need to state the requirements of a system correctly, precisely and completely early in the software development cycle is essential to successful software development. Boehm[2] reports that there are quantitative benefits gained from resolving requirement problems in the early phases of the software life cycle. The later
defects are found in the development cycle, the more expensive is the cost of the software development.

As this initial task of specifying and deriving the requirements drives the later stages of the software development process, it is crucial to establish the requirements correctly and precisely. Establishing the correctness and precision of requirements is difficult: structure may be helpful to enable a better analysis and understanding of the system under development. It is not yet well understood what in structure may be important for the requirements and which techniques may apply.

Structure in requirements may play two roles. First, structure may help in comprehension of requirements. Second, it may help reduce errors when specifying requirements.

In terms of comprehension, as the requirement specification document becomes large, it gets more and more involved and complicated. Such a document is very difficult to understand and analyze to produce good effective software. To understand such a large specification, it is useful to be able to view the structure of the requirements from different points of view.

Structure may also help prevent errors that creep in when a developer writes a specification. In a paper by Betrand Meyer, these errors are referred to as the seven sins of the specifier and are as follows[22, p.7]:

- **Noise:** The presence in the text of an element that does not carry information relevant to any feature of the problem.
- **Silence:** The existence of a feature of the problem that is not covered by any element of the text.
- **Over Specification:** The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
• Contradiction: The presence in the text of two or more elements that define a feature of the system in an incompatible way.

• Ambiguity: The presence in the text of an element that makes it possible to interpret a feature in at least two different ways.

• Forward reference: The presence in the text of an element that uses features of the problem not defined until later in the text.

• Wishful thinking: The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

Some of these errors may be detected by structuring the specification. For example, a part of the specification that shows no relation or dependencies on any other part may be considered as noise. Thus, the structure of the specification helps detect noise. Similarly, it can help detect ambiguity, over specification, forward references, etc.

1.2 Overall Method

This research investigates the concept of structure and structure comprehension in requirement specifications. Two structure comprehension tools, the Software Reflexion Model tool [26] and Rigi[24], that are promising for source code analysis and comprehension, are applied to two sample requirement specifications and the results are analyzed.

The two specifications used were a specification of the telephone registration system (Telereg) of the University of British Columbia[7] and a specification of a Radiation Therapy Machine[15]. Both specifications were written in the formal specification language, Z[31]. Z specifications were chosen because of their formal syntax and the
increasing popularity of the Z language in academia as well as industry. Our goal is to understand what structural concepts are important in the software specification domain and to determine if tools can help identify, comprehend and manipulate the relevant structural concepts.

1.3 The example specifications

1.3.1 The SIS-Telereg system

The Student Information System (SIS) at UBC is designed to establish a single database for student information[7]. It allows decentralized update and inquiry access with appropriate access controls, and provides convenient interfaces to other centralized administrative systems. Support of comprehensive reporting and analysis is also provided as well as the flexibility to implement new policies and to support the changing needs of the University faculties and departments.

Telereg is a telephone registration system, introduced in 1988. It is the student’s interface to the SIS. It allows students to register for their courses from any touch-tone phone. The students use the buttons on the telephone keypad to enter requests and the computer’s voice guides them through their registration. The students may request one or many operations concerning fee, confirmation of registration, etc. Further, details of each section are also provided such as adding a course, dropping a course, changing a course section, confirming a section, switching a section, canceling an entire registration, listing all of a student’s courses and other related requests.

The specification for this system, when written in the formal language Z, offers a precise, comprehensive and accurate description comprising 82 schemas.
1.3.2 The Radiation Therapy Machine

People suffering from certain ailments find radiation therapy quite effective and helpful in alleviating many of their medical problems. The operator's console of a computer controlled radiation therapy machine supports administration of accurate doses of the right kind of radiation at a particular spot[15]. A wrong amount or improper radiation is not only a waste of time and money, more importantly, it might cause irreversible damage to parts of the body and even help in the progress of the disease instead of hindering it. Further some radiation can even cause serious defects and problems in future generations. Thus radiation, which has many benefits, if not administered properly, can be very detrimental. For this reason, it is very important to apply the correct amount of radiation to the right spot.

A radiation therapy machine to be developed in conjunction with the University of Washington Radiation Department was partially formally specified. This formal specification[14], rather than a prose specification, serves as the primary reference source for programming and test planning for the software in the machine. The specified functions include selecting treatment setups from a database of stored prescriptions, setting up prescriptions on the treatment machine manually or semi-automatically, checking that the setup conforms to the prescription, safety interlocking and essential user interface features. This specification supports the concepts of physics and experimental procedures along with patient treatments and consists of approximately 100 schemas.

1.3.3 Why we used these particular specifications

We used these two specifications in our study for the following reasons.
Both specifications are fairly large and cover a wide range of relations and operations. The specifications are written in the formal language, Z. This is a precise specification language that helps the user define the operations and other transformations clearly, precisely, accurately and in detail. Also, using this language, the variables and their types as well as functions can be well understood. This formal language is very similar to a programming language and since both the tools, Software Reflexion Model tool and Rigi, have been designed for programming languages, they can be adapted for the Z environment.

Each specification has many instances of the containment as well as other component which we wanted to study in detail. The different occurrences of these relations give us an opportunity to study the different forms and ways in which they may be used. Furthermore, both of these specifications are very conducive to structuring. Segregations and divisions among the operations are well defined and usually follow a hierarchical pattern.

1.4 The Tools

Both of the tools used, the Software Reflexion Model tool and Rigi are described in Chapter 3. Both of these tools help, in different ways, to investigate the structure of the system. Software Reflexion Model tools form a mapping between the high level model and source code[26]. On the other hand, Rigi uses the concept of reverse engineering in the identification of software artifacts in the subject system and the aggregation of these artifacts to form more abstract system representations[24]. These tools have been applied to source code and have been found useful for program comprehension.
1.5 Thesis Outline

This thesis describes and analyzes the use of two structure comprehension tools, Software Reflexion Models and Rigi, in requirements specification. Chapter 2 discusses the concept of software structure, its importance in programs and how extending this structure to requirement specifications may be beneficial. Chapter 3 describes the two structure comprehension tools, Software Reflexion Model tool and Rigi. The application of the two tools to the two specifications is discussed in chapter 4. Chapter 5 discusses how structure imposed on the two specifications may enhance the understanding of requirements specification. Chapter 6 discusses some of the previous work done in this area and finally, Chapter 7 concludes and summarizes the work.
Chapter 2

Requirement Analysis:
Specification and Structure

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the technical interfaces to people and machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."


2.1 Introduction

As the quote from Brooks above suggests, the most difficult and critical step in software development is the first one, understanding what the project is all about. Software requirement errors have been found to account for a majority of production software failures[4,10], and have been implicated in a number of accidents [20]. Errors introduced during the requirements phase can cost up to 200
times more to correct than errors introduced later in the life cycle[6]. Therefore, techniques to provide adequate requirements specifications and to find errors early are of great importance. The goal is to identify and understand the problem for which a solution is sought and to determine what is to be done. This stage is called the requirement analysis and specification stage.

This stage has two components. The first, requirement analysis, involves stating the problem to be solved and the constraints that exist, on a successful solution to the problem. The second component is requirements specification that involves stating what is expected of the software to be built.

2.2 Requirement Analysis and Specification

2.2.1 Requirement Analysis

One of the most challenging aspects of gathering and specifying requirements is to understand and analyze the sponsor’s problem. Requirement Analysis deals mainly with the following questions[26,p. 45]:

- **Who** is involved in the situation to be analyzed? Who will actually be using the proposed software? Will the users be highly trained individuals, computer novices, or both?

- **What** is the current situation? What is it about the current situation that is posing a problem? What is the proposed system to be? What functions are to be performed by the proposed system?

- **When** must the new system be in place? When will the sponsor be ready to install and test the new system on site?
• **Where** will the new system fit into the old environment?

• **Why** is a new system being sought? Why do users want to change the current system to a new system?

• **How** will the new system function? Are there constraints on how the proposed system can operate?

   Once these important questions have been answered, we are naturally led to the requirements specification stage.

### 2.2.2 Requirements Specification

The requirements specification describes both the functional and non-functional requirements of the system. The functional requirements describe the services that the system is expected to provide by specifying how the inputs to the systems are transformed into outputs. The requirements detail the inputs and their validation as well as the system changes to both normal and error conditions. Non-functional requirements describe the constraints under which the system must operate and any design restrictions imposed on the system, or more specifically, they include details on specific response times, memory requirements, hardware limitations, programming language to be used and any mandatory standards.

The requirements specification describes *what* a system must do as opposed to *how* the tasks must be accomplished. This is an extremely important part of the software lifecycle because it is often seen as a contractual agreement between the customer and the contractor.

Software specifications can be used for different purposes, including as:

• a statement of user needs,
• as a statement of requirements for the implementation, and

• as a reference point during product maintenance.

Above all, the requirements specification is the base document, used for all subsequent developmental activities (Figure 2.1).

![Diagram of software lifecycle](image)

*Figure 2.1: Uses of a Requirement Specification* [7, p.14]

This figure shows the relation of the requirements specification to other of the software lifecycle.

### 2.3 Different Types of Specifications

Many methods can be used to specify requirements in a software system. They differ in terms of expressiveness, features and ease of use. Some of the different methods may be classified as ad-hoc natural language, semi-formal, object-oriented, or formal.
2.3.1 Ad-Hoc Natural Language Specification Approach

This is an informal type of specification that is the most commonly used in industry today. This approach organizes requirements into certain predefined sections, and each section is written in natural language using a free-style format.

Writing requirements in natural language is ad-hoc and open-ended. This approach is subject to ambiguity and imprecision because these properties are inherent in natural language, which is used for the specification.

2.3.2 Semi-Formal Methods

2.3.2.1 Structured Analysis

This is a semi-formal type of specification technique that combines the use of natural language and graphical symbols with some semantics[8]. This methodology describes a system as a function from inputs to outputs. This has a well-defined syntax and loosely defined semantics. Structured Analysis offers a top down specification of the system structured mostly in terms of a set of hierarchical data-flow diagrams.
2.3.3.2 Threads-Based Technique

This is a structured informal specification technique, that is a refinement of the natural language approach. Its basis is the concept of a thread, defined as a “path through a system that connects an external event or stimulus to an output event or response[9]. This technique focuses on external stimuli and responses, and is a method for writing requirements in natural language in a structured manner where the stimuli and responses are made explicit.

This method applies only to the functional requirements section of a system requirements document. It consists of a series of statements, each relating an external stimulus to an external response. The threads-based approach partitions the functional requirements of a system into basic specification units called threads, capabilities and conditions.

2.3.3 Object-Oriented Methods

In object-oriented methods, like Objectory[16] and UML [5], the system requirements are specified in terms of domain objects, their relations and their interactions. In the Objectory method, the different “use cases” of the system are first identified, and the functional requirements for each such case are modeled using a mixture of diagrams and English text. After a series of analysis and transformations, the requirements are evolved into a design specification.
2.3.4 Formal Notation

Webster’s Dictionary defines formal as definite, orderly, and methodical. Thus, to be formal does not necessarily require the use of formal logic, or even mathematics. But in computer science, the phrase “formal methods” has acquired a narrower meaning, referring specifically to the use of a formal notation to represent system models during program development. An even narrower sense refers to the formalism of a method for system development. Typically, first a specification is written in a formal notation and then refined step-by-step into code[21].

Formal methods include the use of mathematical notations such as logic and set theory to describe system specifications and software design together with techniques of validation and verification based on mathematics. These provide very accurate and unambiguous specifications. Hall gives an account of the use of formal methods to eliminate errors:

“In an informal software specification, it is hard to tell what is an error, because it is not clear what is being said. When challenged, people try to defend their informal specification by reinterpreting it to meet the criticism. With a formal specification, we found that errors are much more easily found - and once they are found, everyone is more ready to agree that they are errors.”[12, p. 13]

2.4 Formal methods

The use of formal methods to “write specifications can help minimize errors and can help in precision[21]. Large systems are difficult to develop because they
are conceptually complex and because they are likely to contain errors. Formal specifications give some control over both these problems. First, formal specifications allow the use of abstractions to design large systems, thus reducing their conceptual complexity. Second, formal specifications enable computer-aided design tools to be based on rigorous methods, which can reduce the incidence of errors; this reduces the effort spent on detailed design and implementation of faulty requirements.

Informal specifications, on the other hand, are often ambiguous, requiring direct communication between programmers and designers for clarification. Formal methods are very precise and for this reason they are gaining in popularity. One formal language in use is Z.

2.4.1 Formal Specification Language, Z

One of the most popular formal specification notations that has been used successfully in several industrial applications is the formal specification language, Z[31]. Z is a machine readable mathematical notation with set theory and first order logic as its basis.

A specification written in Z consists of formal mathematical statements that give a precise description of the system. These statements describe the operations performed on data objects. They specify the system state before processing an event (initial state) and after processing the event (final state). This is interspersed with informal text that describes in natural language the meaning of the mathematical statements to make the specification more readable.

Z is used to specify a mathematical model of the system consisting of objects and their relationships. The interactions between the various objects
describe the requirements. The objects are made of elements called basic types. The building blocks for the basic types are sets and functions. Tools are available to provide automatic checking for type consistency across the specification.

Thus, the requirements of a system are described in terms of logic expressions that define sets that describe the data objects of the model, their relationships and the operations performed on them. A Z specification is structured so that it can be developed incrementally and consists of mathematical statements that are packaged together in schemas.

Schemas define states and transitions among the states. The top line of the schema introduces the schema name and the part between this and the dividing line is called the signature. The signature sets out the names and types of the entities used in the schema. The schema predicate (the bottom part of the schema) sets out the relationships between the entities in the signature by defining a predicate over the signature entities.

A schema may be expressed in terms of other schemas and is accompanied by explanatory text that explains the mathematical statements.

For example, consider a schema, PrescribedPatients, in the Radiation Therapy Machine specification: this schema is used to select a prescribed patient for the use of the machine.
This schema includes another schema called SessionVars and describes some operations performed on variables that are defined in SessionVars. The signature is the part above the dividing line and the predicate is the part below the line. The signature part declares the variables used in the schema’s operation and the predicate part describes the operations. The predicate consists of statements. In this example, the predicate has five statements defining the operations of the machine depending on the status of the patient.

Z allows many operations to be performed on the schemas. A few of these are Delta(indicating use of and a change to variables), Xi(indicating use but no change to variables), schema inclusion, schema disjunction, schema conjunction, schema negation, schema composition etc. A detailed description of the Z language may be found in [28].
2.5 **Structuring Requirements**

2.5.1 The Change Process

Any useful software system continually evolves[19]. Changes in such a system are the result of many factors. Some of these may be requests by clients to add more functionality to the system or to improve the maintainability or quality of the code.

To help the software engineer perform the above tasks effectively, it might help the engineer to have some understanding of the structure of the system. In a process that evolves continuously, the software developers and managers must deal with various requests for change. Some of these affect the resulting behavior of the system whereas others affect the non-functional characteristics of the system.

As the structure of the system plays such an important role in the cost of change, information about the structure may be beneficial to an engineer reasoning about a system during the change process [25]. Structuring is a good way of organizing complicated information to make easier to learn. Error detection and correction is typically easier to locate the effects of the error (and thus more errors are located) in a structured system. It is much more difficult to detect omissions or overlaps in a system without structure.

If a system is unstructured, the whole system has to be learnt, a daunting process.

On the other hand, structured systems are easier to update, to test, and to visualize. Structure may also reduce the overall size of the system.

Just as it is the case in programming, structuring may be of great use in writing specifications.
2.5.2 Structure of a System

Requirements specification plays an important part in requirements understanding. Structuring help ease the understanding of the requirements specification by using views, perspectives and abstractions. Views represent partial descriptions of the information system focusing on the requirements of a single group of users[12]. Perspectives specify the requirements concentrating on different aspects and abstractions allow the developer to describe the system at different levels[12]. These three structural mechanisms have mostly been studied in isolation.

The need to support views, perspectives and abstractions has increased for many reasons: to support communications between developers in large projects, to reuse previous artifacts as well as, to organize requirements of legacy systems to re-engineer them. In order to achieve these goals, tools are needed to manage these different aspects and to provide static and dynamic links between them [12].

Like a program, a specification is comprised of various parts or components. These components may be parts of the static view or may also extend to the system’s dynamic view. In each view, interactions, commonly known as relations may occur between the components. These are examples of the aspects of the specification that define the structure of the specification.

One principle that has been adopted in the area of software design that is also important for the field of requirement specification is modularity.

2.5.2.1 Modularity

Modules are the basic building blocks of the software structure. A well-structured system that follows the basic principles of software design should exhibit a logical and
modular structure and support information hiding[8]. Design decisions should be isolated from one another in a way that minimizes the impact of changes in those decisions.

Of the various qualities of software design, none has proven over time to be more significant than modularity. A system can be structured into a collection of modules, where each module has a well-defined interface to other modules, carries out a well-defined function, hides a design decision and can be independently tested and verified.

For modularity, there are several partitioning approaches, including:

- functional decomposition which is based primarily on assigning functions to modules,
- data-oriented decomposition which is based on external data structures,
- event-oriented decomposition which is based primarily on events that the system must handle,
- outside-in design which is based on user inputs to the system, and
- object-oriented design which is based on classes of objects and their interrelationships

One or more of these approaches are typically used in conjunction[23]. Modules should be cohesive, loosely coupled and modules should be black boxes.

In requirements specification as well, modularity is important because:

- duplication of declarations can be avoided by declaring items in only one interface and importing them from there,
- information hiding is encouraged.
- key structural decisions are emphasized, and
- the visibility of structure in the specification is increased.

A requirements engineer wanting to make a change may spend a lot of time sifting through the specification to obtain information to help make the decision. Modularity increases the visibility of the system, so the requirements engineer can spend more time understanding and deciding, rather than gathering, the information on which to base the decision.
2.5.3 Structuring Primitives

Along with modularity, some other primitives have been proposed as structuring mechanisms. The primitives that are more likely to be used in requirements specification include[12]:

1. refinement, view and integration,
2. abstraction, and
3. summarization.

2.5.3.1 Refinements

Refinements allow modeling the same piece of reality in terms of several schemes at different levels of abstraction. The refinement is the typical paradigm used in so called top-down methodologies developed for functional analysis. The different levels are called refinement planes since they refer to the same piece of reality at different levels of abstraction[30].

Refinement planes are a fundamental documentation tool since they allow to perceive a complex reality to be perceived step-by-step, through a generative approach; each step focusing on a limited set of details. In the case of requirements specification, this construct may be explained taking the specification of a birthday book in Z as an example[31]. The birthday book specification records a set of the birth dates, matching the dates with respective names. Further, birth dates may be added, deleted or changed. In such a specification, one schema may depict the overall picture of the birthday book. Another schema that is included in the above schema may detail the mechanism of adding a birthday, yet another may exclusively
concentrate on deleting a birthday, and a third schema on changing a birth date. The specification thus follows a hierarchical structure based on refinement.

2.5.3.2 Abstraction

In computer science, abstraction generally refers to the process of creating a construct that represents selected information in a more compact form than the information itself [31]. When the details of the underlying information are hidden, this process is also referred to as information hiding. In the case of requirements specification, taking the example above, the first schema gives an overall view of the birthday book, including schemas that are described below. These schemas are not defined in the initial schema in which they are included: this is an example of the concept of information hiding.

2.5.3.3 Summarization

"Using summarization, an engineer can flexibly explore the structure of a large system at a reasonable cost. Summarization involves the production of overviews of vast amounts of user-selected information in a timely manner. An overview produced by a summarization process lets detail of the underlying information show through as compared to an overview produced by an abstraction process that hides the underlying detail"[25, p.186].

The technique of summarizing is one commonly used to aid in understanding a problem as well as remembering it. An overview produced by summarization in the case of requirements specification can allow the details of relations to be depicted. For example, in Z, the various relations may include Delta and Xi relations.
We explore, in this research, an appropriate combination of refinement, abstraction and summarization to help understand requirements specification.

2.6 Summary

Specifying requirements is a very important part of the software process. It constitutes analysis of the requirements as well as specifying them accurately and in accordance with the customer's needs. For a reasonably large system, there may be a large number of requirements. Understanding these requirements not only enhances understanding of the system, but also helps in the later stages of software development. Further, maintenance of the requirements is important; it is a necessary aspect of software development. Changes in the software, such as additions and deletions, require a change in the requirements specification. Improving the support for structure in specifications may help ease understanding and maintenance tasks. In this chapter, we have discussed some of the different ways of specifying requirements as well as the constructs used to structure them. The comprehension tools we use to bring out the structure and help in understanding the requirements, and the significance of these tools are discussed in the next chapter.
Chapter 3

Structure Comprehension Tools

"Designing the software structure is the most critical step in creating a good, working software system. The software structure is the framework or skeletal structure of the software system. If the software structure is not well designed, the system will be difficult to build, test and maintain."

3.1 Introduction

It has been suggested that more than 50 percent of software evolution work is devoted to program understanding[34]. Program structure is imperative for program understanding and consequently, maintenance. For the same reasons, it is likely that this argument is also valid for the requirement specification process. Specifying requirements in a structured manner may increase the understandability of the system, which can improve and expedite the process of change, maintenance, and many other activities in software development.
Program structuring has reduced much of the unnecessary complexity of programming. Unlike a spaghetti program, a structured system defines, among other things, a hierarchy among its components: components may be repeatedly nested into larger and larger parts of the system according to the concept of top-down programming.

For large legacy systems, understanding the structural aspects of the entire system’s architecture is typically more important than understanding any one small component. Documentation usually describes isolated parts of the system but not the whole architecture.

The different kinds of the components and relations comprising the source, as well as their distribution over a program makes it difficult for a software engineer to gain an understanding of a system’s structure and also complicates the construction of tools to support the engineer in the understanding process. Another problem faced by both the engineer and the tool provider is the amount and the complexity of structural information that must be considered.

This obvious diversity increases the need for techniques and tools necessary to support an engineer during the “comprehension of the system” process. We need tools that are flexible in handling a variety of structural information and that are also able to scale to support the large amounts of structural information embedded in most software systems.

Based on these observations, various tools have been developed. These tools or techniques may be classified into three categories: program visualization, reverse engineering and (design) conformance tools.

Although, a lot of work has occurred in studying the role of structure in programs, not much work has occurred on the importance of structure in requirements. We study this area and try applying the various ideas used in programming to requirement specifications. The program understanding tools when effectively used in requirement specification may increase understanding, may reduce errors, and may help in building the system
effectively. Consider a large system whose specification constitutes hundreds, in terms of the specification language Z, of schemas. Such a specification would be very difficult, if not impossible, to understand. Using the tools described in the next sections, a graphical representation of the specification can be created that may greatly help understand the specification. The focus of the thesis is on the use of two of these tools (described ahead) in requirement specifications. Are there any differences between their use in programs and requirements specifications? What are the advantages and disadvantages of using these tools for requirement specifications? How can further research be done in this area? As an attempt to answer the above questions, we explore this area. Before describing the study done in this area, we will give a brief overview of the tools and the two example specifications.

3.2 RIGI

The term software visualization is used to describe many different processes. To visually present software artifacts and structure, graphs are particularly suitable. Nodes in the graphs typically represent system components and directed arcs represent the dependencies among these components. However, as the size of software systems increases, so to do their representations as graphs. Advanced graphics and abstraction techniques are needed to manage the visual complexity of these large graphs.

The Rigi reverse engineering system is designed to extract, navigate, analyze and document structure of large software systems[32]. This interactive, visual tool was designed at the University of Victoria to help a software engineer better understand and redocument software.

---

1 Details may be obtained from http://www.rigi.csc.uvic.ca/
The process of reverse engineering is concerned with the analysis of existing software systems so that they are better understood for maintenance, reengineering and evolution purposes[24]. The system is represented in a form where many of its structural and functional characteristics can be visible. This knowledge is then used to improve subsequent development, ease maintenance, re-engineer and aid project management.

The reverse engineering process involves two distinct phases[24]. Parsing the software and storing the extracted artifacts constitutes the first phase. This phase is usually fully automatic. The next phase is typically semi-automatic and helps a developer obtain a model of the structure of the software system. The second phase involves pattern recognition skills and features subsystem composition techniques to generate multiple, layered hierarchies of higher-level abstractions. In this phase, the engineer employs various visualization aids to help recognize patterns, identify candidate subsystem, and to understand software structures in the graph.

The Rigi tool has been developed and used for program understanding, software analysis, reverse engineering and programming-in-the-large. It focuses on querying, representing, visualizing, and evaluating the structure for large, evolving systems. The structure of the system is the organization and interaction between these modules.

The methodology used in Rigi is that of subsystem composition which generates layered hierarchies of subsystems and thereby reduces the complexity of a large specification. Some of the features that promote the use of this tool include:[35]

- an easy-to-use visual interface,

- selection, filtering, and editing operations,

- dependency and change reports,
• standard, overview, and projection perspectives,

• metrics for cohesion and coupling,

• views to capture interesting perspectives,

• scripting language and command library, and

• a customizable user interface.

In an attempt to study and analyze structure in requirement specifications, we applied Rigi to two specifications. One of these, the radiation therapy machine specification, is a reasonably large specification consisting of about a hundred schemas.

In studying Rigi in the context of requirement specification and we were interested to see how it depicts, among other things, the containment relation among schemas. Is it helpful in studying the system in varying levels of detail? Does it help in the understanding of the structure of the system? For example, can this tool help in detecting containment relations? How can requirement components be classified together using this tool? Can this tool be used for a more detailed analysis of the system? If so, how?

### 3.3 Software Reflexion Models

This tool, developed by Gail Murphy at the University of Washington, gives a graphical representation of the high level model of a program and compares it with the source using the relations and interactions between the modules of the source model. This comparison is visualized as a reflexion model[25].

As shown in the Figure 3.1 below, this method basically involves four steps:
- defining a high level model,
- extracting a source model,
- defining a map, and
- computing a reflexion model.

Figure 3.1: Software Reflexion Model[25-p.20]
The reflexion model, basically, provides a comparison between the source model and the high level model.

This cost-effective and lightweight technique can help an engineer explore the structural aspects of a system in various levels of detail. Instead of requiring high-level models to be updated continuously, the software reflexion model technique reduces the risk associated with reasoning in terms of these modules by bridging the gap between the models and the source. Basically, the engineer uses a high-level model as a framework on which to summarize structural information extracted or collected from the software system’s artifacts. The information provided in the reflexion model helps the engineer selectively investigate aspects of the system’s source.

Reflexion Models are used in programs not only to analyze and understand the source code, but also for confirming that a program matches the design. Therefore, they ease the understanding of the system by an analysis of the source code and also help in conforming the program to the system design.

A series of reflexion models may be computed by an engineer to investigate the structure of the system. Thus, the high level model, the mapping and/or the source model may be varied between the computations. As these three inputs are varied, more detail can be added to produce a more detailed reflexion model. Further different ‘views’ may be studied. For example, different levels of detail of the Telereg System were examined and this lead to a more detailed and clearer understanding.

Consider an example which depicts the use of this tool for source code. “An engineer, who has no experience with the NetBSD\textsuperscript{2} implementation is asked to estimate the cost of changing the virtual memory subsystem of the NetBSD Unix operating system to page across a network rather than to a file system, in five days” [25, p.29]. To provide such an estimate, the engineer needs to understand the basic structure of the virtual memory

\textsuperscript{2} NetBSD is a public-domain implementation of a Unix operating system available for a variety of platforms.
subsystem of the NetBSD implementation. This consists of about 250,000 lines of C source code spread over approximately 1900 source files. Due to the complexity and huge amount of code, direct perusal of the code is impossible within the given short time limit. However, the engineer might reason about the task in terms of a high-level model that is based on the engineer's experience with the Unix virtual memory subsystem. Based on such a model, the engineer might reason that the desired change is easy to accomplish by cloning the file system module and migrating it to a module that accesses the network.

The reflexion model technique provides a means of increasing the engineer's confidence in producing estimates for the change task by comparing the earlier model and the structural information extracted from the NetBSD implementation, using the four basic steps outlined earlier. A series of reflexion models may be computed, that describe and study the different levels of detail and the different aspects of the system. A series of reflexion models, computed for the NetBSD virtual memory subsystem provided valuable information about the conformance of the engineer's mental model with the actual structure of the system, thus helping in estimating the cost of changing the virtual memory system to page over a network rather than paging to a file system.

The Software reflexion model may be similarly used in specifications. We used this tool with specifications that use the formal language, Z. Variations in the high level model, depending on the different ways a system may be viewed according to needs and requirements, included in the study. For example, the telereg system is considered from the administrators perspective as well as from the students perspective using high-level models. An overall study of these various views further helps in clarifying the structure of the specification.
3.4 Using Source Code Comprehension Tools for Requirements

Both Rigi and Software Reflexion Model tools may also be used with other kinds of specifications. However, the use of informal specs with tool may be difficult for the reason that the various relations between the components are not stated precisely and therefore, it would be difficult to extract the relations. Other techniques like Data-Flow Diagrams and Structure charts[27] may be used as well with these tools. These particular techniques differ from Z as they are graphical techniques but they use inputs, outputs and various component relations represented by arrows pointing to and from the nodes of the graph, which usually represent components. If these arrows and the nodes of the graph are fed appropriately into the tool, the result would be similar to that when we use the language, Z. However, this might be challenging for the reason that Z is a precise, formal language and thus easier to handle in the required way.

3.5 Experimentation

The work done with the tools is described in detail in the next chapter. Using two specifications, that of Telereg and the Radiation Therapy Machine, we investigate use of the Software Reflexion Model and Rigi tools to display information on the call and containment relation specifically, we specify and highlight the schemas that call another schema.
3.6 Summary

Program understanding is essential in creating an efficient, working software system. Structuring the source code for the system facilitates this process. Some tools have been developed that bring out structure in software. Two of these are the Software Reflexion Model tool and Rigi.

Similarly, understanding the structure of a requirements specification is very important. As we have seen, requirements structure differs from the source code structure.

Our aim is to investigate how source code program comprehension can a software developer tools help comprehend and analyze requirements specification. Such an analysis, along with experiments conducted, the results obtained, and their implications of the results are described in the next chapter.
Chapter 4

Comprehending Software Requirements Specifications

"We fall into error if we attribute to strategy a power independent of tactical results" - Karl Von Clausewitz

4.1 Introduction

Software requirements specification is an early activity in the software cycle that affects all the other development phases as well as the evolution of an artifact. Understanding the software requirements and their dependencies is of paramount importance whether we deal with the initial development or an update of a software system. In this section we investigate how certain tools used for source code comprehension can help in understanding the structure of the software requirements specification when they are specified in a formal language like Z.
In this work, we use two comprehension tools: a conformance tool, the Software Reflexion Model tool, as well as a reverse engineering tool, Rigi. We compare and contrast these two tools using two requirements specifications: the Telereg specification and the Radiation Therapy Machine specification. Various features of both tools are exploited in an attempt to study the structure of the requirements and identify their deficiencies. For instance, we explore how these tools can help to see how one requirement uses another and how certain requirements are grouped together or separated.

The observations made by the tools are described in the following sections.

4.2 Rigi

Initially we used Rigi to examine and understand the two specifications. With Rigi, a developer can produce a layered representation of the specification using the concept of abstraction. The tool allows the users to study the structure of the system, by grouping schemas together as they see fit. Grouping or isolation is done to visualize the structure more clearly. We investigated a parent-child relationship where the child schemas use the parent schemas; more specifically, the child schemas uses some or all of the variables or operations defined in the parent schema. The use of abstraction is intended to facilitate the grouping of schemas into components, which helps in a better understanding of the system.

In the Rigi pictures shown in the following subsection, boxes represent the various components and the arcs represent relations between the components.

For example, a high level view of the telereg system (shown in Figure 4.1) shows that it may be viewed as comprised of four components. Relations between these
components are shown but the contents of these components are hidden. Clicking on one of these boxes (shown in Figure 4.4, for example), another window pops up to show the contents of the component and the interactions between them. This structured representation helps to clarify and understand the system.

4.2.1 The Different Levels

Rigi outputs of the different levels of the telereg system are shown in Figure 4.1 and 4.4-4.7. In these cases, the Telereg system is depicted in a structured manner where the different levels are shown. In the first view a high level model (Figure 4.1) is depicted. Clicking on any one of the "boxes" takes us to a lower level: for example, the contents of the box named sessions and the relations are shown in Figure 4.4. In this manner, lower and lower levels may be studied and a clear picture of the structure of the system may be obtained as in Figure 4.8.

To obtain the initial boxes, the user needs to collapse related schemas manually and label them appropriately. Such grouping is usually based on some component property. The user may choose an attribute and use that to collapse components with that attribute into one unit. Once this collapsing is done, the box may be expanded by clicking on it. This process of collapse and expansion results in a simplified view of the structure of the system. In this study, we used Rigi with two requirements specifications written in Z. We were particularly interested in three relations, Delta, Xi and Include, that are common to most Z specifications. We studied these relations using features of Rigi to isolate one kind of relation from the other.

We say that a schema A uses another schema B, if B appears in the predicate of A. In this case A and B are the first and second operands of the use relation. In the picture we
see an arrow going from A to B. We also say that schema A uses a delta (or Xi) instance of schema B if B appears in a delta (or Xi) expression in the signature of A. In this case, A is the first and B is the second operand of the delta (or Xi) relation. To show these relations, we need to define the appropriate source model for the specification.

Schemas related semantically are grouped together. For example, all the schemas involved in the processing of an application are clustered together in a box called Processing.

Similarly, all operations in the Telereg system pertaining to logging in were put together in a box called Logging-in. Actions involving registration and fee checks are in PreProcessing and all the definitions that are used in the specification are in Definitions. An entire operation, say if a user wants to change a section he/she is registered in, requires schemas from more than one box. For instance, the Telereg operations in PreProcessing would be used to check whether the user is a valid registered student, then the system would allow the user to log in after which the user may process his/her request by entering the box called sections, ongoing into Processing. Full details on operations require following a downward course through the various levels by choosing the appropriate box in each section.
We are mainly interested in two types of relations, the call relation and the Delta/Xi relation, between the Z schemas. A part of the source model used for Rigi is shown in Figure 4.2-4.3. There are different files for the call relation and the Delta/Xi relation. Each of the files consists of three columns. The first column describes the type of the relation. The second column gives the name of the first operand schema, whereas the third column names the second operand schema. The specification of the LSME patterns[26] to extract the source model is attached in Appendix A.
Call ISTTS Telereg State
Call IRS Telereg State
Call TAO Telereg State
Call MCNE Telereg State
Call SEIS Telereg State

*Figure 4.2: the source model for the call relation*

delta ATCL Telereg State
delta RFCL Telereg State
delta DO Telereg State
delta IO Telereg State
delta ATS Telereg State

*Figure 4.3: the source model for the delta relation*

The line starts at bottom of the box that uses the schema and ends at the top of box where the schema is defined. As explained before, Figure 4.1 is a view of the system at the highest level. It shows that the Telereg system is composed of four components and that 'Definitions' is used in all the other components 'Processing', 'PreProcessing' and 'Logging-In'.
Figure 4.4: Level 2

Figure 4.4 shows the second level of the Telereg system. It consists of five independent components. These components contain the specifications of the various requests and are as follows:

- **sections**: the processing of a request to add a section, drop a section, and change a section.

- **eert**: execute exit request thread. This box consists of two schemas involved in exiting Telereg. One schema defines the request and the other outlines the response.
• hhup: handle hangup process. This box consists of a schema that handles the conclusion of a Telereg session, that is, hanging up the phone system.

• cftnc: checks if fee for transaction must be charged

• cvrc: check valid request condition. Checks if the request is valid and if an appropriate amount of fee has been paid.

Figure 4.5: Level 3
Component “sections”, which is at the third level, is comprised of six boxes: ‘loist’, ‘lact’, ‘ast’, ‘csst’, ‘aast’ and ‘cst’. All of these operations involve a course in a section. Clicking on ‘aast’ takes us to a still deeper level (shown in Figure 4.6), while picture 4.7 shows the components of ‘aast’ which is at the lowest level of this specification.

Further, rlmt has two parts: RIRR(RLMT_input_repeat_request) and RRF.

*Figure 4.6: Level 4*
The path:
processing -> sections -> aast -> rlmt
takes a user through one “thread” in the specification. A thread in the specification is a
series of actions which lead us from the highest level to the lowest level.

So far we have shown how the different levels of a specification can be observed separately using Rigi. Additionally, a complete picture of the structure of the entire system
may be obtained using a feature of Rigi’s called Simple Hierarchical Multi-Perspective (SHriMP) views.\textsuperscript{3}

Figure 4.8: The entire ‘structured’ telereg specification

\textsuperscript{3} This information was obtained in a conversation with Margaret Anne Storey. Information about this feature of Rigi may be obtained from [24]
Such a view is shown in Figure 4.8. Each block is one 'view' in Rigi. Clicking on the required block gives details of that box (i.e., a lower level of the system). In our examples, only the first block in each level was shown in detail.

This overall view of the system is important because it highlights the overall structure. Thus using Rigi, an engineer may obtain a high-level view of the system or may conveniently click on a part of the requirement specification and obtain the details of that particular area.

To realize the importance of structuring, we tried to study the 'unstructured' specification (Figure 4.9) and compare it to the structured specification. The unstructured view is just a collection of boxes that represent the schemas arranged randomly. As the reader can see, it is quite messy even for small specifications and it is almost impossible to understand and figure out the structure of the system. This makes the design and implementation of the system a challenging task.
Similar work was done with the radiation therapy machine specification, which is a much more elaborate specification. The results are shown in Figures 4.10 to 4.14. Figures 4.10 to 4.13 show the structure of some nodes at various levels while Figure 4.14 shows the whole specification without any structuring.
The radiation therapy machine specification at the highest level consists of six units/boxes; that is, this specification is divided into the following six parts at the highest level:

- Session
- User_Interface
- System_Configuration
- Field

Figure 4.20: Level 1
• Combining_the_subsystems

• Software_interlocks

Here again, the blocks were grouped semantically. The operations used for grouping were those used in the specifications. Using the Z language, these operations were Delta, Xi, and includes schemas. All schemas involving a particular kind of operations were grouped together. The prescriptions for the patients are all in one block called 'Prescription_Database'. The entire session of the machine is in 'Session_state' and all the operations performed are in a block called 'Operations'.
On going into "session", we find that it consists of three boxes:

- Prescription_Database,
- Session_state, and
- Operations.

Figure 4.11: Level 2
Prescription_Database may be further divided into:

- Field_state,
- Login, and
- Confirm.

*Figure 4.12: Level 3*
On going into "confirm", we find five further states:

- ConfirmOp,
- MenuOp,
- ME,
- MSC, and
- GMA.
In contrast, a view of the unstructured specification is shown in Figure 4.14.

Clearly, a structured specification shows more promise to be easier to understand and implement than an unstructured one. A structured specification shows the different levels of the requirements and specifies the interactions within a level and between the levels.

In the following sequence, we study some other aspects that are important for requirements specifications. First we explore how a user can isolate a specific relation
between the specification components and how the tools can be used to show the component layout with respect to that relation. Figure 4.15 shows the telereg specification with the delta relation only.

![Diagram showing the Delta relation in the Telereg specification](image)

**Figure 4.15: The Delta relation in the Telereg specification**

In Figure 4.15, the lines between the different components show that one component (the component where the line starts at the bottom of the box) uses the other component (the one where the line ends at the top of the box) in a Delta relation. In Z, this indicates that there is a transition between the first and a second components.
Similarly, in the radiation therapy machine specification, we isolated several relations. Figure 4.16 shows the “include” schema relation. Figure 4.17 shows the Xi relation indicating a transition relation between the initial and final component of a schema where the initial component is equal to the final component. Finally, Figure 4.18 shows the Delta relation.

![Diagram of General - 1 Root <<ACTIVE>>](image)

*Figure 4.16: The call/include relation - Radiation Therapy Machine*
Isolating a relation helps the user determine the manner in which components use other components. Thus, for example when a change is made in a component we can determine which components will be affected and in what way. Further, it may show that certain components do not interact in certain ways and that they may be isolated. We can also note that not all schemas participate in all kinds of relations with other schemas.
We also investigate whether a different criteria for classification can be useful. Using the syntactic rules of Z, we grouped together all schemas that included other schemas in a certain way. Schemas involving the Delta relation were all in one group, schemas involving the Xi relation were in another and a third group consisted of schemas that use another in its predicate. The result for Telereg is shown in Figure 4.19.
Figure 4.19: Telereg specification viewed syntactically

The boxes in Figure 4.19 represent the different types of interactions in the specification and clicking on a particular box gives details of that interaction only (as done in the manual isolation of the relations in the specification). For example, Figure 4.20 depicts only the boxes involved in the Include (or call) relation.
4.3 Software Reflexion Model

As it was indicated in the previous chapter, the Software Reflexion Model tool produces a reflexion model that provides a comparison between a high level model and a source model of the system. This tool may prove useful for specifications as it checks the accuracy of the structure of the requirements specifications if the engineer has some idea of the structure. We studied the scope and flexibility of the tool and various operations.
allowed by the tool. Different levels of detail of the specification were observed which helped us to get a clear picture, not only of the structure of the system at the highest level, but also of the inner depth of the system. Further, the different types of relations in the specification are studied in isolation and their contribution in determining the structure of the system is studied. For each study performed, we present a brief explanation and show the reflexion model produced. The reflexion models of the two example specifications, the telereg specification and the radio therapy machine specification, are shown in Figures 4.21 and 4.22.

![Diagram](image-url)

**Figure 4.21: Reflexion Model of Telereg Specification**
In both the cases, we chose a level that showed enough detail to understand the structure of the specification, but was not too involved or complicated.

In these models, the boxes represent the components of the specification and the arrows show the relations between the components. The number on the arrows depicts the number of times one component uses another. The solid lines show the convergences (where the source model agrees with the high level model). The dashed arrows are the divergences (where arcs not predicted by the high-level model appear in the source model) and the dotted lines are the absences (where the source model does not include arcs predicted by the high level model)[25]. The reflexion model of the Telereg specification
shows that this specification consists of four components, ‘LoggingIn’, ‘Preprocessing’, ‘Processing’ and ‘Definitions’. Each component is involved in a group of related operations. Three of these components use the fourth and this is depicted by the arrows. Further ‘Definitions’ uses itself 19 times by defining a variable that is later used in another definition.

In order to obtain this model, we wrote a source file, a mapping file, and a high level model file. On inputting these files into the tool, the reflexion model is obtained. Details on the language used to write these files may be obtained from [25]. Appendices B and C show the mapping and high-level model used for Telereg.

Using the reflexion model, knowledge of the structure of the system is obtained. A similar reflexion model is shown for the radiation therapy machine specification. It can be seen that this specification is much more complicated and the model is even more useful in this case. Absences in the source model, that are predicted in the high level model, are shown in the Software Reflexion Model and, therefore; a user can determine what needs to be added to a source model in order to complete it. A complex and involved specification is harder to understand. The Software Reflexion Model tool can give a simplified high-level view and enable a detailed study of the relations and interactions at different levels, allowing us to study parts of the specification as well as to obtain its overall structure.

4.1.1 Observations and their Implication

Having an appropriate high-level model is very important in the Reflexion Model technique. We first used the Telereg specification and changed the high-level model and the source model input file, and found some interesting results.
Different levels or views of the telereg specification are observed. First a very high-level model was used. The Telereg system was divided into two parts, one that is accessed directly by the students and the other contained the administrative requirements (Figure 4.23).

As this did not give us too much useful information about the specification, we changed the model to give a more detailed view (one step lower). The Student operations was divided into more detail and the reflexion model, shown in Figure 4.24, was obtained.

To get a clearer picture, a yet more detailed view was obtained. Thus, in any specification, for a more and more detailed view it is possible to refine the reflexion model till the desired detail is achieved. We divided each of the sections, ‘LoggingIn’, ‘PreProcessing’, and ‘Processing’ to obtain more detail and we obtained a still deeper level.
that gives the following structure. These refinements help identify the structure of the system. As more and more detail is studied, the hierarchical structure of the system is observed in more depth.
Figure 4.25: The most detailed specification of the Telereg system. As expected there are no relations between telereg_state and global_types.
These different refinements give a picture of the structure of the specification and help ease the understanding of the specification. For example, in the Telereg specification, on studying the three refinement levels, an idea of the structure of the specification can be formed. We can deduce that at the highest level there are two components, that of Students and that of Definitions.

Students may be further divided into three kinds of processes.

- processes which concern logging into the system (LoggingIn)
- preprocessing (PreProcessing) and
- finally, processing the required function (Processing).

These three components may further broken down.

LoggingIn consists of

axiomatic_definitions.

- PreProcessing consists of WSRC (Withhold student From Registration Condition) and VRR (Validate Registration Request).

Processing consists of five parts:

- fee(fee),
- Execute Exit Request thread(EERT),
- sections(sections),
- Add Specialization Thread(AST), and
• Handle hangup condition (HHUP).

Analyzing the specification using such high-level models can reveal some interesting properties. By studying the include, Delta and Xi relations we can see that each substructure ("block") depends on only one "block", telereg_state. This conclusion helps in identification of the structure of the system and will help in the later stages of software development.

*The Typed Reflexion Model:*

There is a feature in the reflexion model tool to differentiate between the various types of relations. Such a ‘typed’ reflexion model can help isolate a particular relation. Isolation and detection of these relations is essential in studying the structure of the system.

Figure 4.26 shows the typed reflexion model for the Telereg specification. In this figure, the two arrows going from one box to another represent the “Delta” and the “Includes” relation. In Figure 4.26, the lines on the left hand side between each pair of boxes represents the “Delta” relation, whereas the lines on the right hand side represents the “Includes” relation.

*Figure 4.26: The Telereg System - Typed Reflexion Model*
A typed Reflexion Model of the Radio Therapy machine was also studied and is shown in Figure 4.27. As the Radio Therapy Machine specification is more complicated, we can see that the typed reflexion model helps in gaining an understanding of the specification as different layers of the specification can be separated. The top layer, which is less complicated, gives a clear view of the overall structure of the system. The isolation of the different interactions and relations enables us to study each one independently and in detail, providing a clear picture of the structure of the system.

This specification is very involved and complicated and isolation simplified it.
Figure 4.27: Typed Reflexion Model - Radiation Therapy Machine
Figure 4.27 shows three different relations used in this specification. The relations may be isolated by using three different figures instead of this one. Different figures for each prove useful in a complicated specification where many relations shown together may result in confusion.

Further, we thought that different criteria may be used to group together the various schemas and thus a clearer understanding of the structure of the specification may be obtained. For example the schemas may be grouped syntactically: all the schemas with the “Delta” relation may be grouped together and are represented by the box labeled Delta and all those with the “includes” relation form another cluster (the box labeled Include). The schemas which are members of these two clusters may be put in another cluster (box labeled Common). The arc is from “Delta” to “Telereg_State” shows that there are 41 schemas that have the interaction “Delta Telereg_State”, meaning they involve a change in “Telereg_State”. Although this was a very good summarization of the specification based on syntactic properties and allowed us to observe the different relations and their interactions, we found that this syntactic view does not help in identifying the actual structure of this particular specification.

![Figure 4.28](image.png)
4.3 Software Reflexion Model and Rigi: A comparison

Although the Rigi and the Reflexion model tools are fundamentally different techniques, our study showed that both can be used to highlight the structure of specifications. A high-level model is created of the specification and in this way both of these techniques can help in gaining an understanding of the system. Various relations and the interactions between the modules can be identified.

Both of these techniques provide insight to the structure of the specification, though in different ways. One difference is that the Software Reflexion Model tool is a design conformance tool whereas Rigi is a reverse engineering tool.

Another difference is the views supported by the techniques. The Software Reflexion Model technique shows the interactions between different modules. Different levels of the specification can be viewed by changing the source model and the high-level model (refinement). In the case of Rigi, module interactions are used to build a detailed, layered model of the specification. Different levels may be viewed by clicking on the required subsystem.

The way in which views are created also differs. The Software Reflexion Model uses a combination of top-down and bottom-up processes. This helps this tool handle complexity and scale in the source model. However, Rigi uses the top down process for structuring the specification.

Using Rigi, an engineer can express the desired clustering of requirement entities, either manually or automatically based on the source model. The same can be done with the Reflexion Model tool.

As a final example, in the case of Rigi, even when significant clustering has been performed to derive a high-level model, this model may not necessarily comprise of a view
of interest to the engineer. The entire system participates in this view. A particular view may be isolated using domains in this tool[24]. However, in the case of the Software Reflexion Model, a particular relation may be selected and the entire specification may be viewed with respect to that relation only.

Using these tools, the specifier can easily identify the components of a specification. For this purpose, the Software Reflexion Model tool requires the use of a source model, a high-level model and a mapping file to produce a pictorial, easy-to-understand representation of the specification. Rigi produces a pictorial representation from the source model. The software reflexion model tool needs an intuitive picture of the high level of the specification and information on how the detailed parts map to the high level. It produces a reflexion model that shows whether the high level model is a correct representation of the specification.

4.3 Summary

Requirements specifications are different from source code as requirements may have different structure as compared to the structure of a program. For instance, the requirements components (or units) may be different from the traditional source code modules, and relations between the components may be different as well. Nevertheless, there is a similar need to structure the requirements as to structure programs as a way of comprehending their structure. The use of a formal specification technique greatly enhances that possibility. We have shown that program comprehension tools can be used with requirements specified in the formal language, Z, in a similar way to how they are used with source code.

The results show that these two tools can be effective in gaining an understanding of a specification. Different levels of the system are understood by using refinement in the
Software Reflexion Models and by studying the different levels of abstraction in Rigi. We see that both these tools help in an identification of the structure in requirements.

The process of modularity helps identify and separate the different functions performed in the specification. Modularity helps in the simpler process of understanding parts of the system separately which may then be combined to give the whole system. We found these two tools were effective in identifying the structure of requirement specifications. This structure recognition of the specification may be extremely beneficial to ease comprehension and thus reduce errors in system development.
Structured Specifications

“In the specification notation known as Z, schemas are used to structure mathematical descriptions”

5.1 Introduction

We have seen that structure in specifications can help in understanding, and possibly maintaining, the system. In this chapter, we show how Z can be extended to include a grouping (or scope) primitive. This extension provides an explicit means of specifying structure similar to packages or modules in modern programming languages. Structuring allowed us to break the system into many smaller components. We chose to collect the “semantically grouped” components and study each in isolation. These components can be further arranged in different levels. For example, components that “use/contain” schemas can be in one level and the schemas they “use” can be at a lower level.
Certain languages may be more conducive to structuring as compared to others. For example, these formal constructs have unambiguous semantics which cannot be violated by the user.

A structuring an extension for Z can help the user define a clear picture of the task each component definition. A component definition is no more than a higher level of scoping or naming space mechanism than that which exists in many programming languages like Ada and C++.

There are many ways to add component definitions to Z. We have chosen a notation similar to that which appears on the first line of the definition.

A typical component, with an embedded component, would thus be expressed as in Figure 5.1.

```
*******example**************************************
 |
 |
 |
 |
 |
 variable : set
 variable : set

 ********************Uses*************************
 |
 |
 |
 |
 |
 A------------------------------------------
 |
 |
 |
 |
 |
 |   ********************embedded_layer********
 |
 |
 |
 |
 |
 |
 |
 |
 |
 variable : set
 variable : set
 A------------------------------------------
 |
 |
   ********************embedded_layer********
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
the component. It ends with the last line and contains a number of declarations and other components. For better examples, the reader may look at the structured schemas of the two example specifications, the Telereg system and the Radiation Therapy Machine, which are attached in the appendix D and E, respectively.

A specification can be divided into various modules depending on functionality or other criteria. Together or all schemas involving a Delta relation may be grouped together. Here we consider semantic differences only.

Forcing the designer to make the requirements specification more structured may not only ease understanding of the system, thus making software development easier, but also may help in detecting errors and easing the maintenance of the requirements in the long run.

Finally, the component definition can be used as a convenient scoping mechanism. We can use the same name in different components to mean different things. The above example uses two different schemas, both called A, in two different subsections. By adding a scoping operator ::, we can distinguish between these. One is referred to as Uses::A whereas the other is referred to as embedded _layer::A.

5.2 The two structured specifications

In this sections we provide some explanation about the structuring of the two example specifications used in the thesis using the new Z concept.
5.2.1 The Telereg specification

Consider the telereg specification. Using the new construct, the specification is divided into four sections: Definitions, LoggingIn, PreProcessing and Processing. Each is further divided into smaller parts. The number of layers and the position of a schema may be identified by counting the lines on the left-hand side or right-hand side. For example, consider the construct Try_Again_Request_Condition. This box is comprised of two schemas: TARC_maximum_timeouts_exceeded and TARC_register_request_valid_condition. We can easily see that it is in the third layer. This entire structured specification is shown in appendix D.

5.2.2 The Radiation Therapy Machine specification

The radiation therapy machine specification is divided into six sections: Session, User-interface, System-Configuration, Field, Combining_the_subsystems, and Software_Interlocks. Again each section is further divided into smaller parts. The number of layers and the position of a schema may be identified by counting the lines on the left-hand side or right-hand side. For example consider the construct Session. This section is further divided into five sections. Each section contains one or more schemas. This entire structured specification is shown in appendix E.

5.2 Summary

Because of the importance of structure in specifications, an attempt is made to force structure in specifications from the initial stage. For this purpose, certain features are
added to the specification language, Z, that allow the user to structure the specification into components or sections. The two example specifications are written including these structural features to depict how these features help in understanding the specification.
Chapter 6

Related Work

Much work has been done on structuring in source code, but not much work has been done on structuring requirements and on tools for requirements specification. Some concepts for structuring requirements are discussed by Francalanci et al [12]. They describe three dimensions of requirements engineering, focusing on structuring requirements using views, perspectives and abstractions. They claim that usually all the artifacts produced during requirements specifications should be organised using these structured dimensions. Efforts are being made to provide methods and tools that allow the organization of requirements combining all these dimensions. Further they discuss the concept of reuse in requirements and how a structured specification supports reuse more than an unstructured specification. Various tools that allow control of relations in a specification are studied and their features are outlined. We used the different structuring approaches in our study. Using Rigi and the Software Reflexion Model, different views of
the two specifications are studied. Using Rigi, we are able to look at the different levels of both systems.

Another approach that concentrates on the use of views in Z specifications is presented in a paper by Daniel Jackson[17]. A view may be described as a partial specification of a program, consisting of a state space and a set of operations. The full specification may be obtained by composing several views and linking them through their states and through their operations. View structuring lends clarity and terseness to the specification as it encourages multiple representations of a program’s state.

Jackson claims that the structure of most of the published Z specifications mostly follows the structure of an implementation [17]. Even though at this low-level the structures diverge (predicates over sets replace loops, pointers and so on), the gross organisation often retains the flavor of a program: the global variables are placed in a common area and the operations are packaged into modules.

As a view is a partial specification of the entire system, it may be evaluated directly against the perceived requirements and can be constructed and analyzed independently of other views. Thus, the complexities of the dependencies may be deferred until a later stage, when the views are connected. This makes a view specification easier to construct and maintain. Features of Z, such as implicit preconditions, explicit frame conditions and uniform set representation, are conducive to this technique of view structuring[17]. In other languages, pervasive use of invariants and redundancy, that is used in Z, is not encouraged. For this reason, view structuring is rarely seen in V DM or Larch. We used detailed descriptions of the dependencies between schemas in a particular view to help structure a specification. Our views were imposed on existing specifications rather than being used in constructing a specification. Different kinds of views may be necessary depending n whether a developer is trying to construct or understand a specification.
Another construct that may be used with the Z notation is that of box structures[11].

This method proposes developing a two stage specification: a top level black-box specification followed by a state-box specification. The black-box and the state box describe identical external behavior. The difference is that the black-box behavior is described using stimulus histories, whereas the state box behavior is described by referencing the internal system state. Before we split the two specifications we studied into detailed views, we roughly estimated the boundaries of the black box and the state box. This helped in the detailed analysis of the specification.
"Unlike a mechanical engineer who can often take a device apart and see how the parts fit together, a software engineer is faced to comprehend systems that are pure thought-stuff and infinitely malleable."

Specifications have little explicit structure. This lack of structure can make it difficult to understand and evolve the specification. For example, in particular it can be difficult for a software engineer to understand the affected parts of a specification when a change is made to a specific group of requirements.

The same problems appeared in programs long ago and forced the language designers to provide explicit structuring mechanisms. For instance package/module concepts were introduced in Modula3 and Ada to manage complexity and limit the effects of change.

The introduction of structure in programs greatly helped to reduce errors and avoid redundancies. As requirement specifications are even more basic than programs and as they are the first and probably most crucial part of the software life cycle, the introduction of structure at this level may prove even more beneficial. This study investigated the use of the existing structure determination techniques written for source code to the problem of
determining structure in requirement specifications. More specifically we studied how Software Reflexion Models and Rigi can be used with requirements specified using the Z specification language.

We conducted various experiments on these tools to study their functionality in some special cases. Their features were examined and studied in the context of requirements specification.

The concepts of modularity, abstraction and refinement allow a programmer to break down a large, complex program into smaller and more manageable parts. We found that these concepts also helped to clarify understanding of requirements specifications. We used particular specifications that are written in a formal language, Z. Being mathematical and precise in nature, such specifications can be easily structured in layers or hierarchies and thus can lead to an understanding of the system.

To demonstrate this point, the two specifications used were structured in a variety of ways and the benefits of such structuring were discussed. We saw that Rigi and Software Reflexion Models were helpful in detecting structure in the specifications and understanding the specification.

Using the Software Reflexion Model and Rigi, we divided each of the two example specifications into a number of basic components and studied the interactions between them. The Software Reflexion Models compared the high level model with one we envisioned and specified the number of convergences, divergences and absences with our model. In the case of Software Reflexion Model among other things, the typed reflexion model and refinement proved very useful.

We found that the collapse-expand feature of Rigi helped in abstraction and Shrimp Views proved very useful in obtaining a comprehensive picture of the entire specification.

The use of both these tools provided us with an accurate description of the specification.

We also discussed a high-level structuring mechanism for Z that allows designers to
structure their specifications from the start, when specifying the requirements, rather than reconstructing structure after the requirements are complete.

7.1 Future Work

The structure determination techniques used may be explored further and more functionality may be added to them. Presently, work is being done on Rigi to provide an interface that may give a view of the entire program structure (Shrimp Views) as well as the substructure which would allow the parent and children nodes may be viewed together. In the Software Reflexion Model, some work might be done in the context of requirement specifications. There may be provision to extend this technique to the other different kinds of specification, some of which involve diagrammatic techniques. Other extensions to this technique may include techniques to reduce the size of the mapping file and such extension have been described by Gail Murphy[25] in the context of programming languages.
Bibliography


APPENDIX A
(Source patterns extracting information from the Telereg specification)

schema %
( - )+ \(<\text{schemaname}> ( - )+\)
%
schema.delta %
[ \| ]\![\text{Delta}<\text{deltavar}> @
write("&1&\text{schemaname}, &2&"," &1&\text{deltavar}, &2&")
@
%
schema.xi %
[ \| ]\![\text{Xi}<\text{xivar}> @
write("&1&\text{schemaname}, &2&"," &1&\text{xivar}, &2&")
@
%
schema.include %
\| <\text{newvar}> \| @
write ("&1&\text{schemaname}, &2& &1&\text{newvar}, &2&")
@
%
(The mapping file for the Software Reflexion Model for the Telereg)

```plaintext
[ schema=Telereg_State mapTo=Definitions ]
[ schema=InSTTSchedule mapTo=Definitions ]
[ schema=InRegularSchedule mapTo=Definitions ]
[ schema=AddToClassList mapTo=Definitions ]
[ schema=RemoveFromClassList mapTo=Definitions ]
[ schema=DecrementOpenings mapTo=Definitions ]
[ schema=IncrementOpenings mapTo=Definitions ]
[ schema=AddToSchedule mapTo=Definitions ]
[ schema=RemoveFromSchedule mapTo=Definitions ]
[ schema=ThereAreOpenings mapTo=Definitions ]
[ schema=MaxCreditsNotExceeded mapTo=Definitions ]
[ schema=StudentEnrolledInSection mapTo=Definitions ]
[ schema=WithdrawStudentFromSection mapTo=Definitions ]
[ schema=RegisterStudentInSection mapTo=Definitions ]
[ schema=Wsrc_Holds_Or_Blocks_Blocks_Condition mapTo=LoggingIn ]
[ schema=Cofac_Outstanding_Fee_Condition mapTo=LoggingIn ]
[ schema=Vit_* mapTo=LoggingIn ]
[ schema=Ccfat_* mapTo=LoggingIn ]
[ schema=Cadc_* mapTo=PreProcessing ]
[ schema=Cerc_* mapTo=PreProcessing ]
[ schema=Prt_* mapTo=PreProcessing ]
[ schema=Cert_* mapTo=PreProcessing ]
[ schema=Irrc_* mapTo=PreProcessing ]
[ schema=Tarc_* mapTo=PreProcessing ]
[ schema=Cftnc_* mapTo=Processing ]
[ schema=Cvrc_* mapTo=Processing ]
[ schema=Eert_* mapTo=Processing ]
[ schema=Rlmt_* mapTo=Processing ]
[ schema=Act_* mapTo=Processing ]
[ schema=Csst_* mapTo=Processing ]
[ schema=Cst_* mapTo=Processing ]
[ schema=Dst_* mapTo=Processing ]
[ schema=Loist_* mapTo=Processing ]
[ schema=Lact_* mapTo=Processing ]
[ schema=Hhup_* mapTo=Processing ]
[ schema=Hhup_* mapTo=Processing ]
```
APPENDIX C

The high level model for the Software Réflexion Model for the telereg specification

LoggingIn Definitions
PreProcessing Definitions
Processing Definitions
APPENDIX D

TELEREG - STRUCTURED SPECIFICATION

specification

*********Definitions**********************************************

*********Global_Types**********************************************

[Session]
[Course]
[Balance]
[Specialization]

RegistartionType ::= STT_Schedule | Regular
Schedule == P Section
Openings == Z
Classlist == P Sid
Credits == Z
Counter == Z
Free_Transactions == Z
LockModes ::= LOCKED | UNLOCKED

Register_Requests ::= ADD | DROP | LOOKUP | CHANGE | EXIT | CONFIRM | LIST | CANCEL | ADDSPEC

Stimuli_Or_Responses ::= ...
add_section_msg | add_section_req<<Section>> | add_spec_msg<<Specialization>> | add_spec_req<<Specialization>> | ask_spec_msg |
bye_msg |
cancel_msg |
cancel_rejection |
cancel_req. |
cannot_add_msg |
cannot_change_msg |
cannot_confirm_msg |
cannot_drop_msg |
cannot_register_msg |
change_msg |
change_req<<Section>> |
confirm_msg |
confirm_req<<Section>> |
drop_msg |
drop_req<<Section>> |
exit_msg<<Balance>> |
exit_req<<Session>> |
fee_charged_msg |
fee_kickout_msg |
fee_msg<<Balance x Session>> |
fee_request<<Session>> |
free_transactions_left_msg<<Counter>> |
hangup |
hold_or_block_msg |
invalid_id_msg |
invalid_request_msg<<Register_Requests>> |
invalid_spec_msg |
last_name_msg |
list_msg<<Course>>
list_req | lookup_msg<<Openings>> | lookup_reg<<Section>> | maximum_charge_fee_msg | request_id_msg | repeat_reg | say_spec_msg<<Specialization>> | session<<Session>> | session_msg<<Session>> | student_id<<SID>> | try_again_msg

System_State ::= Waiting_for_identification_input | Waiting_for_session_input | Waiting_for_spec_input | Waiting_for_next_request_input | IDLE

>= _ : Balance <-> Balance

<_ _ : Date <-> Date

Telereg_State:

Students : P SID
RegisteredIn : SID -> Schedule
Availability : Section -> Openings
Class : Section -> Classlist
ScheduleType : SID -> RegistrationType
Blocks : P SID
Holds : P SID
Outstandings : P SID
ValidSession : P Session
Eligibles : SID -> Session
Accessed : SID -> Date
Account : SID -> Balance
LockStatus : SID -> LockModes
ChargeFeeStatus : SID -> Balance
state : System:State
count : Free:Transactions
StudentSpec : SID -> Specialization
ValidSpec : P Specialization
MAX:TIMEOUT:COUNT : Z
MAX:ID:COUNT : Z
max:credits : Z
MAX:CHARGE:FEE := Balance
Commands : P Register:Requests
CostlyCommands : P Register:Requests
NumberCredits : Section -> Z
TotalCredits : SID -> Z
CourseSchedule : Section -> Course

(dom RegisteredIn) subseteq Students
Blocks subseteq Students
Holds subseteq Students
Outstandings subseteq dom Account
Outstandings subseteq Students
CostlyCommands subseteq Commands

Axiomatic_Definitions
---RemoveFromSchedule---

Delta Telereg_State
id : Sid
section : Section
newschedule : Schedule

newschedule = RegisteredIn id \n    (section)
RegisteredIn' = RegisteredIn +
    (id |-> newschedule )

---ThereAreOpenings---

Telereg_State
section : Section

Availability section > 0

---MaxCreditsNotExceeded---

Telereg_State
id : Sid
section : Section

TotalCredits id + NumberCredits
section <= max_credits

---StudentEnrolledInSection---

Telereg_State
id : Sid
section : Section

section in (RegisteredIn id)

---WithdrawStudentFromSection---

Delta Telereg_State
id : Sid
section : Section
newschedule : Schedule
newclasslist : Classlist
newOpenings : Openings

RemoveFromSchedule
RemoveFromClassList
IncrementOpenings

---RegisterStudentInSection---

Delta Telereg_State
id : Sid
section : Section
newschedule : Schedule
newclasslist : Classlist
newOpenings : Openings

AddToSchedule
AddToClassList
DecrementOpenings

Logging In

---WSRC_Holds_or_blocks_condition---
VIT_Requirement1 =
VIT_input_student_identification and 
VIT_in_identification_input_state and 
not VIT_id_found_condition and 
VIT_max_attempt_exceeded_condition 
implies VIT_connection_rejection and 
VIT_move_to_idle_state

VIT_Requirement2 =
VIT_input_student_identification and 
VIT_in_identification_input_state and 
not VIT_id_found_condition and 
not VIT_max_attempt_exceeded_condition 
implies 
VIT_invalid_identification_rejection 
and VIT_identification_request

VIT_Requirement3 =
VIT_input_student_identification and 
VIT_in_identification_input_state and 
VIT_id_found_condition and 
VIT_withhold_condition 
implies 
VIT_hold_or_block_rejection and 
VIT_move_to_idle_state

VIT_Requirement4 =
VIT_input_student_identification and 
VIT_in_identification_input_state and 
VIT_id_found_condition and 
VIT_outstanding_fee_condition implies 
VIT_outstanding_fee_rejection

VIT_Requirement5 =
VIT_input_student_identification and 
VIT_in_identification_input_state and 
VIT_id_found_condition and 
not VIT_withhold_condition implies 
VIT_student_identification_acceptance 
and VIT_move_to_session_input_state

Validate_Identification_Thread =
VIT_Requirement1 and 
VIT_Requirement2 and 
VIT_Requirement3 and 
VIT_Requirement4 and 
VIT_Requirement5

*******************************************************************************
***Compute_Current_Fee_Assesment_Thread**********

-CCFAT_input_fee_balance_request------

Telereg_State
ss : Session
input? : Stimuli_Or_Responses

input? = fee_request(ss)

-CCFAT_fee_balance_feedback----------

Delta Telereg_State
id : Sid
ss : Session
output! : Stimuli_Or_Responses

output! = fee_msg
(Account id), ss)
Telereg_State
id : Sid
id in Holds or id in Blocks

******Fee**********

---COFAC_outstanding_fee_condition---
Telereg_State
id : Sid
id in Outstandings

*****Validate_Identification_Thread************

--VIT_input_student_identification---
Telereg_State
id : Sid
input? : Stimuli_Or_Responses
output! = last_name_msg

-VIT_invalid_identification_rejection---
Delta Telereg_State
output! : Stimuli_Or_Responses
output! = invalid_id_message

-VIT_connection_rejection---------
Delta Telereg_State
output! : Stimuli_Or_Responses
output! = bye_msg

-VIT_hold_or_block_rejection---------
Delta Telereg_State
output! : Stimuli_Or_Responses
output! = hold_or_block_msg
id in Students

-VIT_max_attempt_exceeded_condition---
Telereg_State
id_count : Z
id_count >= MAX_ID_COUNT

VIT_withhold_condition =^ WSRC_holds_or_blocks_condition
VIT_outstanding_fee_condition =^ COFAC_outstanding_fee_condition
VIT_in_identification_input_state =^ [Telereg_State | state = Waiting_for_identification_input]
VIT_move_to_idle_state =^ [Delta Telereg_State | state' = IDLE]
CCFAT_in_session_input_state = \{\text{Telereg\_State} \mid \text{state} = \text{Waiting\_for\_session\_input}\}

CCFAT_in_next_request_input_state = \{\text{Telereg\_State} \mid \text{state} = \text{Waiting\_for\_next\_request\_input}\}

Compute\_Current\_Fee\_Assessment\_Thread

CCFAT_input_fee_balance_request and
CCFAT_in_next_request_input_state or
CCFAT_in_session_input_state implies
CCFAT_fee_balance_feedback

-----------------------------------------------

*****Preprocessing**********************************

*****Registration************************************

*****Check\_Access\_Dates\_Condition*************

-CADC_no_access_condition-----------

\begin{align*}
\text{Telereg\_State} \\
\text{ss : Session} \\
\text{currentDate : Date} \\
\text{id : Sid} \\
\text{currentDate < Accessed id}
\end{align*}

-----------------------------------------------

*****Check\_Eligibility\_To\_Register\_Condition*****

-CERC_no_eligibility_condition-------

\begin{align*}
\text{Telereg\_State} \\
\text{ss : Session} \\
\text{id : Sid} \\
\text{(id \rightarrow ss) subseteq Eligibles}
\end{align*}

-----------------------------------------------

*****Preprocess\_Registration\_Thread**************

-PRT_input_session---------------------

\begin{align*}
\text{Telereg\_State} \\
\text{ss : Session} \\
\text{input? : Stimuli\_Or\_Responses} \\
\text{input? = session(ss)}
\end{align*}

-PRT_session_feedback-----------------

\begin{align*}
\Delta \text{Telereg\_State} \\
\text{ss : Session} \\
\text{output! : Stimuli\_Or\_Responses} \\
\text{output! = session\_msg(ss)}
\end{align*}

-PRT_register_attempt_rejection------

\begin{align*}
\Delta \text{Telereg\_State} \\
\text{output! : Stimuli\_Or\_Responses} \\
\text{output! = cannot\_register\_msg}
\end{align*}
\(-\text{delta_telereg\_state}\) 

\(\text{output!} : \text{Stimuli\_Or\_Responses} \) 

\(\text{output!} = \text{ask\_spec\_msg} (\text{StudentSpec\ id}) \) 

\(-\text{PRT\_session\_valid\_condition}\) 

\(\text{Telereg\_State} \) 
\(\text{ss} : \text{Session} \) 

\(\text{ss} \text{ in ValidSession} \) 

\(-\text{PRT\_need\_specialization\_condition}\) 

\(\text{Telereg\_State} \) 
\(\text{id} : \text{SId} \) 

\(\text{not (forall spec : Specialization} \) 
\(\text{\@ {id |-> spec} subseteq} \) 
\(\text{StudentSpec}) \) 

\(\text{PRT\_move\_to\_spec\_input\_state} =^= \) 
\(\text{[Delta Telereg\_State | state' = Waiting\_for\_spec\_input]} \) 

\(\text{PRT\_move\_to\_idle\_state} =^= \) 
\(\text{[Delta Telereg\_State | state' = IDLE]} \) 

\(\text{PRT\_move\_to\_next\_request\_state} =^= \) 
\(\text{[Delta Telereg\_State | state' = Waiting\_for\_next\_request\_input]} \) 

\(\text{PRT\_Requirement1} =^= \) 
\(\text{PRT\_input\_session and} \) 
\(\text{PRT\_in\_session\_input\_state and} \) 
\(\text{PRT\_session\_valid\_condition and} \) 
\(\text{PRT\_no\_access\_condition or} \) 
\(\text{PRT\_no\_eligibility\_condition} \) 
\(\text{implies} \) 
\(\text{PRT\_register\_attempt\_rejection and} \) 
\(\text{PRT\_move\_to\_idle\_state} \) 

\(\text{PRT\_Requirement2} =^= \) 
\(\text{PRT\_input\_session and} \) 
\(\text{PRT\_in\_session\_input\_state and} \) 
\(\text{PRT\_session\_valid\_condition and} \) 
\(\text{not PRT\_no\_access\_condition and} \) 
\(\text{not PRT\_no\_eligibility\_condition and} \) 
\(\text{not PRT\_need\_specialization\_condition} \) 
\(\text{implies} \) 
\(\text{PRT\_session\_feedback and} \) 
\(\text{PRT\_specialization\_feedback and} \) 
\(\text{PRT\_move\_to\_next\_request\_input\_state} \) 

\(\text{PRT\_Requirement3} =^= \) 
\(\text{PRT\_input\_session and} \) 
\(\text{PRT\_in\_session\_input\_state and} \) 
\(\text{PRT\_session\_valid\_condition and} \) 
\(\text{not PRT\_no\_access\_condition and} \) 
\(\text{not PRT\_no\_eligibility\_condition and} \) 
\(\text{PRT\_move\_to\_spec\_input\_state} \) 

\(\text{Preprocess\_Registration\_Thread} =^= \) 
\(\text{PRT\_Requirement1 and} \) 
\(\text{PRT\_Requirement2 and} \) 
\(\text{PRT\_Requirement3} \)
**Cancel_Entire_Registration_Thread**

- **CERT_input_cancel_all**
  
  \[
  \text{Telereg\_State} \\
  \text{input? : Stimuli\_Or\_Responses} \\
  \text{input? = cancel\_req}
  \]

- **CERT_cancel_all_feedback**
  
  \[
  \text{Delta Telereg\_State} \\
  \text{output! : Stimuli\_Or\_Responses} \\
  \text{output! = cancel\_msg}
  \]

- **CERT_cancel_all_rejection**
  
  \[
  \text{Delta Telereg\_State} \\
  \text{output! : Stimuli\_Or\_Responses} \\
  \text{output! = cancel\_rejection}
  \]

\[
\text{CERT_in\_next\_request\_input\_state} = \text{state = Waiting\_for\_next\_request\_input} \\
\text{CERT_valid\_request\_capability} = \text{Check\_Validity\_Of\_Requests\_Capability} \\
\text{CERT_outstanding\_fee\_condition} = \text{COFAC\_outstanding\_fee\_condition} \\
\]

- **CERT_remove_each_section**
  
  \[
  \text{Delta Telereg\_State} \\
  \text{id : SID} \\
  \text{newclasslist : Classlist} \\
  \text{newopenings : Openings} \\
  \text{newschedule : Schedule} \\
  \]
  
  \[
  \text{forall section : Section} @ \\
  \text{Student\_Enrolled\_In\_Section and} \\
  \text{Withdraw\_Student\_From\_Section}
  \]

\[
\text{CERT\_Requirement1} = \text{CERT\_input\_cancel\_all and} \\
\text{CERT\_valid\_request\_capability and} \\
\text{CERT\_in\_next\_request\_input\_state and} \\
\text{not CERT\_outstanding\_fee\_condition and} \\
\text{not InSTTSchedule implies} \\
\text{CERT\_remove\_each\_section and CERT\_cancel\_all\_feedback}
\]

\[
\text{CERT\_Requirement2} = \text{CERT\_input\_cancel\_all and} \\
\text{not CERT\_valid\_request\_capability or} \\
\text{not CERT\_in\_next\_request\_input\_state or} \\
\text{CERT\_outstanding\_fee\_condition or} \\
\text{InSTTSchedule implies} \\
\text{CERT\_cancel\_all\_rejection}
\]

\[
\text{Cancel\_Entire\_Registration\_Thread} = \text{CERT\_Requirement1 and CERT\_Requirement2}
\]
Telereg_State
request_count : Z
request_count > MAX_TIMEOUT_COUNT

-IRRC_register_request_valid_condition--
Telereg_State
request : Register_Requests
request in Commands

-IRRC_concurrent_access_condition----
Telereg_State
id : SID
LockStatus id = LOCKED

IRRC_request_rejection_condition =\nIRRC_concurrent_access_condition or
not IRRC_register_request_valid_condition
and IRRC_maximum_timeouts_exceeded

********Try_Again_Request_Condition***********

-TARC_maximum_timeouts_exceeded------
Telereg_State
request_count : Z
request_count > MAX_TIMEOUT_COUNT

-TARC_register_request_valid_condition--
Telereg_State
request : Register_Requests
request in Commands

TARC_try_again_condition =\nTARC_register_request_valid_condition and
not TARC_maximum_timeouts_exceeded

********Processing***********************

******Charge_fee_For_Transaction_If_Necessary_Condition*****

-CFTNC_charge_transaction_fee_condition---------
Telereg_State
request : Register_Requests
output! : Stimuli_Or_Responses
output! * try_again_msg

*******CheckValidityOfRequestsCapability***********

-CVRC_free_transactions_left_feedback----------
Delta Telereg_State
transact_count : Counter
output! : Stimuli_Or_Responses

output! = free_transactions_left_msg (transact_count)

-CVRC_fee_charge_feedback-----

Delta Telereg_State
output! : Stimuli_Or_Responses

output! = fee_charged_msg

-CVRC_maximum-charge_fee_rejection-----

Delta Telereg_State
output! : Stimuli_Or_Responses

output! = maximum_charge_fee_msg

-CVRC_register_request_valid_condition-----

Telereg_State
request : Register_Requests

request in Commands

CVRC_charge_transaction_fee_condition >= CFTNC_charge_transaction_fee_condition

CVRC_request_rejection_condition >= IRRC_request_rejection_condition

CVRC_try_again_condition >= TARC_try_again_condition

CVRC_less_than_ten_free_transactions_condition-----

Telereg_State

count <= 10

-CVRC_maximum_charge_fee_exceeded-----

Telereg_State
id : SID

ChargeFeeStatus id >= MAX_CHARGE_FEE

CVRC_move_to_idle_state = [Delta Telereg_State | state' = IDLE ]

-CVRC_decrement_fee_transactions-----

Delta Telereg_State

count' = count - 1

CVRC_Requirement1 =
CVRC_request_rejection_condition implies invalid_request_rejection and
CVRC_move_to_idle_state

CVRC_Requirement2 =
CVRC_try_again_condition implies CVRC_try_again_request
and:

\[ \text{CVRC} \_\text{less} \_\text{than} \_\text{ten} \_\text{free} \_\text{transactions} \_\text{condition} \implies \text{CVRC} \_\text{free} \_\text{transactions} \_\text{left} \_\text{feedback} \]

\[ \text{CVRC} \_\text{Requirement}4 \equiv \]
\[ \text{CVRC} \_\text{register} \_\text{request} \_\text{valid} \_\text{condition} \\text{and} \]
\[ \text{CVRC} \_\text{maximum} \_\text{charge} \_\text{fee} \_\text{exceeded} \implies \]
\[ \text{CVRC} \_\text{maximum} \_\text{charge} \_\text{fee} \_\text{rejection} \\text{and} \]
\[ \text{CVRC} \_\text{move} \_\text{to} \_\text{idle} \_\text{state} \]

\[ \text{CVRC} \_\text{Requirement}5 \equiv \]
\[ \text{CVRC} \_\text{register} \_\text{request} \_\text{valid} \_\text{condition} \\text{and} \]
\[ \text{not} \\text{CVRC} \_\text{maximum} \_\text{charge} \_\text{fee} \_\text{exceeded} \\text{and} \]
\[ \text{CVRC} \_\text{charge} \_\text{transaction} \_\text{fee} \_\text{condition} \implies \]
\[ \text{CVRC} \_\text{fee} \_\text{charge} \_\text{feedback} \\text{and} \]
\[ \text{CVRC} \_\text{decrement} \_\text{free} \_\text{transactions} \]

\[ \text{Check} \_\text{validity} \_\text{of} \_\text{requests} \_\text{capability} \equiv \]
\[ \text{CVRC} \_\text{Requirement}1 \\text{and} \]
\[ \text{CVRC} \_\text{Requirement}2 \\text{and} \]
\[ \text{CVRC} \_\text{Requirement}3 \\text{and} \]
\[ \text{CVRC} \_\text{Requirement}4 \\text{and} \]
\[ \text{CVRC} \_\text{Requirement}5 \]

execute Exit Request Thread

---

**EERT_input_exit_request**

```
| Telereg_State |
| ss : Session |
| input? : Stimuli_Or_Responses |
```

```
| input? = exit_req(ss) |
```

---

**EERT_exit_feedback**

```
| Delta Telereg_State |
| id : Sid |
| ss : Session |
| output!: Stimuli_Or_Responses |
```

```
| output! = exit_msg(Account id) |
```

---

**EERT_in_next_request_input_state**

```
| [Telereg_State | state = Waiting_for_next_request_input] |
```

**EERT_valid_request_capability**

```
| Check_Viability_Of_Requests_Capability |
```

**EERT_move_to_idle_state**

```
| [ Delta Telereg_State | state’ = IDLE ] |
```

**Execute.Exit_Request_Thread**

```
| EERT_input_exit_request and |
| EERT_valid_request_capability and |
| EERT_in_next_request_input_state implies |
| EERT_exit_feedback and |
| EERT_move_to_idle_state |
```

---

**Sections**

---

**Repeat_Last_Message_Thread**

---

```
| -RLMT_input_repeat_request--
| Telereg_State |
| input? : Stimuli_Or_Responses |
```

```
| input? = repeat_req |
```
ACT_Requirement1 and
ACT_Requirement2

*******Confirm_A_Section_Switch_Thread**************

-CSST_input_confirm_switch-----------------------------------
  Telereg_State
  section : Section
  input? : Stimuli_Or_Responses
  input? = confirm_req(section)

-CSST_confirm_switch_feedback-----------------------------------
  Delta Telereg_State
  output! : Stimuli_Or_Responses
  output! = confirm_msg

-CSST_confirm_switch_rejection-----------------------------------
  Delta Telereg_State
  output! : Stimuli_Or_Responses
  output! = cannot_confirm_msg

CSST_in_next_request_input_state ^=
  [Telereg_State | state = Waiting_for_next_request_input]

CSST_valid_request_capability ^=
  Check_Validity_Of_Requests_Capability

CSST_outstanding_fee_condition ^=
  COFAC_outstanding_fee_condition

CSST_Request1 ^=
  CSST_input_confirm_switch and
  CSST_valid_request_capability and
  CSST_in_next_request_input_state and
  not CSST_outstanding_fee_condition and
  not InSTTschedule and
  StudentEnrolledInSection implies
  CSST_confirm_switch_feedback

CSST_Request2 ^=
  CSST_input_confirm_switch and
  not CSST_valid_request_capability or
  not CSST_in_next_request_input_state or
  CSST_outstanding_fee_condition or
  InSTTschedule or
  not StudentEnrolledInSection implies
  CSST_confirm_switch_rejection

Confirm_A_Section_Switch_Thread ^=
  CSST_Request1 and
  CSST_Request2

*******Change_Section_Thread*******************************

-CSST_input_change_section-----------------------------------
  Telereg_State
  section : Section
  input? : Stimuli_Or_Responses
  input? = change_req(section)
-RLMT_repeat_feedback-----------------------------
Delta Telereg_State
lastmsg : Stimuli_Or_Responses
output! : Stimuli_Or_Responses

output! = repeat_msg(lastmsg)

Repeat_Last_Message_Thread ^=
RLMT_input_repeat_request implies
RLMT_repeat_feedback

-------------Add_A_Section_Thread-------------------

- ACT_input_add_section----------------------------
Delta Telereg_State
section : Section
input? : Stimuli_Or_Responses

input? = add_section_req(section)

- ACT_add_section_feedback-------------------------
Delta Telereg_State
output! : Stimuli_Or_Responses

output! = add_section_msg

- ACT_add_section_rejection------------------------
Delta Telereg_State
output! : Stimuli_Or_Responses

output! = cannot_add_msg

ACT_in_next_request_input_state ^=
[Telereg_State | state = Waiting_for_next_request_input]

ACT_valid_request_capability ^=
Check_Validity_Of_Requests_Capability

ACT_outstanding_fee_condition ^=
COFAC_outstanding_fee_condition

ACT_Requirement1 ^=
ACT_input_add_section and
ACT_in_next_request_input_state and
ACT_valid_request Capability and
not ACT_outstanding_fee_condition and
not InSTTSchedule and
ThereAreOpenings and
MaxCreditsNotExceeded and
not StudentEnrolledInSection implies
RegisterStudentInSection and
ACT_add_section_feedback

ACT_Requirement2 ^=
ACT_input_add_section and
not ACT_in_next_request_input_state or
not ACT_valid_request_capability or
ACT_outstanding_fee_condition or
InSTTSchedule or
not ThereAreOpenings or
not MaxCreditsNotExceeded or
StudentEnrolledInSection implies
ACT_add_section_rejection
-CST_change_section_feedback---------

Delta Telereg_State
output! : Stimuli_Or_Responses

---------
output! = change_msg

-CST_change_section_rejection------

Delta Telereg_State
output! : Stimuli_Or_Responses

---------
output! = cannot_change_msg

CST_in_next_request_input_state =^=
[Telereg_State | state = Waiting_for_next_request_input]

CST_valid_request_capability =^=
CheckValidity_Of_Requests_Capability

-CST_old_section_found_condition-----

Telereg_State
section : Section
id : Sid

---------
exists old_section :
Section @ [CourseSchedule
old_section = CourseSchedule
section and (old_section in
Registeredln id)]

CST_outstanding_fee_condition =^=
COPAC_outstanding_fee_condition

CST_Requirement1 =^=
CST_input_change_section and
CST_valid_request_capability and
CST_in_next_request_input_state and
not CST_outstanding_fee_condition and
not InSTTSchedule and
ThereAreOpenings and
CST_old_section_found_condition and
not StudentEnrolledInSection implies
RegisterStudentInSection and
WithdrawStudentFromSection[old_section
/section] and CST_change_section_feedback

CST_Requirement2 =^=
CST_input_change_section and
not CST_valid_request_capability or
not CST_in_next_request_input_state or
CST_outstanding_fee_condition or
InSTTSchedule or
not ThereAreOpenings or
not CST_old_section_found_condition or
StudentEnrolledInSection implies
CST_change_section_rejection

Change_Section_Thread =^=
CST_Requirement1 and
CST_Requirement2

******************************************************************************

*****Drop_Section_Thread***********************

-DST_input_drop_section--------------

Telereg_State
section : Section
input? = drop_req(section)

-DST_drop_section_feedback-------

Delta Telereg_State
output! : Stimuli_Or_Responses

output! = drop_msg

-DST_drop_section_rejection-------

Delta Telereg_State
output! : Stimuli_Or_Responses

output! = cannot_drop_msg

DST_in_next_request_input_state =^=
[Telereg_State | state = Waiting_for_next_request_input]

DST_valid_request_capability =^=
Check_Validitiy_Of_Requests_Capability

DST_outstanding_fee_condition =^=
COPAC_outstanding_fee_condition

DST_Requirement1 =^=
DST_input_drop_section and
DST_valid_requestCapability and
DST_in_next_request_input_state and
not DST_outstanding_fee_condition and
not InSTTSchedule and
StudentEnrolledInSection implies
WithdrawStudentFromSection and
DST_drop_section_feedback

DST_Requirement2 =^=
DST_input_drop_section or
not DST_valid_request_capability or
not DST_in_next_request_input_state or
DST_outstanding_fee_condition or
InSTTSchedule or
not StudentEnrolledInSection implies
DST_drop_section_rejection

Drop_Section_Thread =^=
DST_Requirement1 and
DST_Requirement2

***Look_Up_Inquiry_On_Sections_Thread****

-LOIST_input_lookup_section-------

Telereg_State
section : Section
input? : Stimuli_Or_Responses

input? = lookup_req(section)

-LOIST_lookup_section_feedback------

Delta Telereg_State
section : Section
output! : Stimuli_Or_Responses

output! =
lookup_msg(Availability section)
**LIST_in_next_request_input_state**

\[ \text{LIST_in_next_request_input_state} =^= \{ \text{Telereg_State} \mid \text{state} = \text{Waiting_for_next_request_input} \} \]

**LIST_valid_request_capability**

\[ \text{LIST_valid_request_capability} =^= \text{Check Validity Of Requests Capability} \]

**Look_Up_Inquiry_Or_Sections_Thread**

\[ \text{Look_Up_Inquiry_Or_Sections_Thread} =^= \text{LIST_input_lookup_section and} \]
\[ \text{LIST_valid_request_capability and} \]
\[ \text{LIST_in_next_request_input_state implies} \]
\[ \text{LIST_lookup_section_feedback} \]

---

**List_All_Courses_Thread**

**LACT_input_list_all**

\[ \text{LACT_input_list_all} \]

\[ \text{Telereg_State} \]
\[ \text{input?} \ast \text{Stimuli_Or_Responses} \]
\[ \text{input?} = \text{list_req} \]

**LACT_list_all_feedback**

\[ \text{Delta Telereg_State} \]
\[ \text{id} : \text{Sid} \]
\[ \text{output!} : \text{Stimuli_Or_Responses} \]
\[ \text{forall } s : \text{Section @ S in} \]
\[ (\text{RegisteredIn id}) \text{ and} \]
\[ \text{output!} = \text{list_msg} \]
\[ (\text{CourseSchedule s}) \]

**LACT_in_next_request_input_state**

\[ \text{LACT_in_next_request_input_state} =^= \{ \text{Telereg_State} \mid \text{state} = \text{Waiting_for_next_request_input} \} \]

**LACT_valid_request_capability**

\[ \text{LACT_valid_request_capability} =^= \text{Check Validity Of Requests Capability} \]

**LACT_outstanding_fee_condition**

\[ \text{LACT_outstanding_fee_condition} =^= \text{COFAC_outstanding_fee_condition} \]

**List_All_Courses_Thread**

\[ \text{List_All_Courses_Thread} =^= \text{LACT_input_list_all and} \]
\[ \text{LACT_valid_request_capability and} \]
\[ \text{LACT_in_next_request_input_state and} \]
\[ \text{not LACT_outstanding_fee_condition implies} \]
\[ \text{LACT_list_all_feedback} \]

---

**Add_Specialization_Thread**

**AST_input_add_specialization**

\[ \text{AST_input_add_specialization} \]

\[ \text{Telereg_State} \]
\[ \text{spec :Specialization} \]
\[ \text{input?} \ast \text{Stimuli_Or_Responses} \]
\[ \text{input?} = \text{add_spec_req(spec)} \]

**AST_add_specialization_feedback**

\[ \text{Delta Telereg_State} \]
\[ \text{id} : \text{Sid} \]
\[ \text{output!} : \text{Stimuli_Or_Responses} \]

---

168
**Handle Hangup Process**

```
HHUP_hangup_condition

Telereg_State
| input? : Stimuli_Or_Responses
| input? = hangup
```
HHUP_move_to_idle_state =^=
[ Delta Telereg_State | state' = IDLE. ]

Handle_Hang_Up_Process =^=
HHUP_hangup_condition implies
HHUP_move_to_idle_state

end specification
APPENDIX E

RADIATION THERAPY MACHINE - STRUCTURED SPECIFICATION

************************************************************

***********System_configuration******************************

***********Settings_and_registers******************************

ITEM := nfrac | dose_tot | dose | wedge | w_rot | filter |
        | leaf0 | leaf39 | gantry | collim | turnt | lat | longit |
        | height | dose5 | top | pt_mode | pt_factor | press | temp |
d_rate | t_fac  | calvol1 | calvol2 | p_dose | p_time | e_time |

| setting, dose_reg : P ITEM

<setting, dose_reg> partition ITEM

dose_reg = { pt_mode, pt_factor, press, temp, d_rate,
t_fac, calvol1, calvol2, p_dose, p_time, e_time }

scale, selection, counter : P ITEM

<selection, scale, counter> partition ITEM

counter = { nfrac, dose_tot, dose } 

selection = { wedge, w_rot, filter, pt_mode } 

leaves = (leaf0, leaf39) 

preset == leaves union { wedge, w_rot, filter } 

motion == preset union { gantry, collim, turnt, lat, longit,
height } 

dprescrip == motion union counter 

dprescrip = { lat, longit, height } 

sensor == setting \ { nfrac, dose_tot } 

cal_const == { d_rate, t_fac, calvol1, calvol2 } 

******************************************************************************

*******Values**************

VALUE == Z

blank : VALUE

tol : SCALE --> value

valid : ITEM --> FLVALUE

for all s : ITEM @ blank notin valid s

******************************************************************************

*****Interlocks**************

INTLK :: = clear | set

INTERLOCK ::= collision | gc_cc | tmc_error | fw_cc |
            | fw_fault | lc_cc | lcc_cal | lcc_error | 
            | dosim | pt_intlk | dmc_cal | setup_timeout |
            | start_timeout | dmc_error | door | console |
            | pedestal | operator | update | plc_error |
            | cs_error | proton |

******************************************************************************
**Subsystems**

\[
\text{SUBSYS ::= gantry\_couch | filters | collimator | dosimetry | intlks | proton\_beam,}
\]

**settings**

\[
\begin{align*}
\text{gantry\_couch} & \rightarrow \{ \text{gantry, collim, turnt, lat, longit, height, top} \}, \\
\text{filters} & \rightarrow \{ \text{wedge, w\_rot, filter} \}, \\
\text{collimator} & \rightarrow \{ \text{leaf0, leaf39} \}, \\
\text{dosimetry} & \rightarrow \{ \text{nfrac, dose\_tot, dose, doseB} \}, \\
\text{intlks} & \rightarrow \{ \}, \\
\text{proton\_beam} & \rightarrow \{ \}
\end{align*}
\]

**interlocks**

\[
\begin{align*}
\text{gantry\_couch} & \rightarrow \{ \text{collision, gc\_cc, tmc\_error} \}, \\
\text{filters} & \rightarrow \{ \text{tmc\_error, fw\_cc, fw\_fault} \}, \\
\text{collimator} & \rightarrow \{ \text{lcc\_cc, lcc\_cal, lcc\_error} \}, \\
\text{dosimetry} & \rightarrow \{ \text{dosim, pt\_intlk, dmc\_cal, setup\_timeout, start\_timeout, dmc\_error} \}, \\
\text{intlks} & \rightarrow \{ \text{door, console, pedestal, operator, update, plc\_error, ca\_error} \}, \\
\text{proton\_beam} & \rightarrow \{ \text{proton} \}
\end{align*}
\]

**Prescription_database**

\[
\begin{align*}
\text{[ NAME ]}
\end{align*}
\]

\[
\text{PATIENT} = \text{NAME}; \text{FIELD} = \text{NAME}
\]

\[
\begin{align*}
\text{no\_name} = \text{NAME}
\end{align*}
\]

\[
\begin{align*}
\text{no\_patient} = \text{no\_name} ; \text{no\_field} = \text{no\_name}
\end{align*}
\]

\[
\begin{align*}
\text{studies, patients} : P \text{ PATIENT}
\end{align*}
\]

\[
\begin{align*}
\text{no\_patient} \text{ notin studies and no\_patient} \text{ notin patients}
\end{align*}
\]

**ACCUMULATION**

\[
\begin{align*}
\text{counter} \rightarrow \text{VALUE}
\end{align*}
\]

**PRESCRIPTION**

\[
\begin{align*}
\text{prescrip} \rightarrow \text{VALUE}
\end{align*}
\]

\[
\begin{align*}
\text{Preset} : \text{studies} \rightarrow (\text{FIELD} \rightarrow \text{PRESCRIPTION}) \\
\text{Prescribed} : \text{patients} \rightarrow (\text{FIELD} \rightarrow \text{PRESCRIPTION}) \\
\text{Accumulated} : \text{patients} \rightarrow (\text{FIELD} \rightarrow \text{ACCUMULATION})
\end{align*}
\]

\[
\begin{align*}
\text{forall } s : \text{studies @ no\_field notin } \text{dom(Preset } s \text{ )}
\end{align*}
\]

\[
\begin{align*}
\text{forall } p : \text{patients @ no\_field notin } \text{dom(Prescribed } p \text{ ) and dom(Prescribed } p \text{ ) = dom(Accumulated } p \text{ )}
\end{align*}
\]

**exceeded**

\[
\begin{align*}
\text{ACCUMULATION} \leftrightarrow \text{PRESCRIPTION}
\end{align*}
\]

\[
\begin{align*}
\text{forall } \text{counters : ACCUMULATION ; fields : PRESCRIPTION @ exceeded} (\text{counters, fields}) \leftrightarrow (\text{exists } c : \text{counter @ counters } c \geq \text{ fields =})
\end{align*}
\]

**Operators**

\[
[\text{OPERATOR}]
\]
no_operator : OPERATOR
operators, physicists : P. OPERATOR

\[ \text{physicists \subseteq operators} \]

******************************************************************************

******Session**********************************************

******Session_State*********************************************

MODE ::= therapy | experiment

---SessionVars---
mode : MODE
operator : OPERATOR
patient : PATIENT
field : FIELD
names : P PATIENT
fields : FIELD -> PRESCRIPTION
counters : FIELD -> ACCUMULATION

---operator = no_operator or operator in operators
.mode = experiment implies operator in physicists
names = if mode = therapy then patients else studies

---NoPatient---
patient = no_patient
field = no_field
fields = ()
counters = ()

---PrescribedPatient---
patient /= no_patient
patient in names
field = no_field or field in domain fields
fields = if mode = therapy then
Prescribed patient
Preset patient
mode = therapy implies counters = Accumulated patient

Session = Fresh PrescribedPatient or NoPatient

---InitSession---
NoPatient
mode = therapy
operator = no_operator

***********************************************************************

******Operations******************************************************

---ExptModeS---
Delta Session

operator in physicists
NoPatient
(mode', names') = if mode = therapy then
(experiment, studies) else (therapy, patients)

operator' = operator
StoreFields

Delta Session

field? : FIELD
prescribed' : PRESCRIPTION
accumulated' : ACCUMULATION

\[
\text{patient} /= \text{no}_\text{patient} \\
\text{field}' = \text{field} \\
\text{fields}' = \text{fields} \cup \\
\{\text{field}' \rightarrow \text{prescribed}'\} \\
\text{mode} = \text{therapy} \implies \text{counters}' = \\
\text{counters} \cup \{\text{field}' \rightarrow \text{accumulated}'\} \\
\text{mode}' = \text{mode} \\
\text{operator}' = \text{operator} \\
\text{patient}' = \text{patient} \\
\text{names}' = \text{names}
\]

***Login***

---NewOperator---

Delta Session

operator? : OPERATOR

operator' = operator?
operator' in operators

---Privileged---

NewOperator

mode = therapy or operator' in physicists

mode' = mode
patient' = patient
names' = names
field' = field
fields' = fields
counters' = counters

---Unprivileged---

NewOperator

mode = experiment
operator' notin physicists
mode' = therapy
NoPatient'

LoginS = Privileged or Unprivileged

---SelectPatient---

Delta Session

patient? : PATIENT

\[
\text{patient} in \text{names} \\
\text{patient}' = \text{patient} \\
\text{field}' = \text{no}_\text{field} \\
\text{fields}' = \text{if mode} = \text{therapy} \text{then Prescribed patient}' \\
\text{else Preset patient}' \\
\text{mode} = \text{therapy} \implies \text{counters}' = \\
\text{Accumulated patient}' \\
\text{mode}' = \text{mode} \\
\text{operator}' = \text{operator} \\
\text{names}' = \text{names}
\]
---SelectFields---
Delta Session
field? : FIELD

patient \neq \text{no\_patient}
field? in \text{dom\_fields}
field' = field?
operator' = operator
mode' = mode
patient' = patient
fields' = fields
counters' = counters

---Field---

```plaintext
**Field_state**

\text{cal\_factor}, \text{cal\_const} \rightarrow \text{VALUE}

**Field**

\text{prescribed : PRESCRIPTION}
\text{accumulated : ACCUMULATION}
\text{measured : sensor \rightarrow VALUE}
\text{overridden : prescrip \rightarrow VALUE}
\text{computed, calibrated : dose\_reg \rightarrow VALUE}
```

---Relation_to_Session_state---

**PrescribedField**

Field
Session

field \neq \text{no\_field}
mode = therapy implies
prescribed = fields field

no_prescrip = (\lambda p : \text{prescrip} @ \text{blank})
no_counter = (\lambda c : \text{counter} @ \text{blank})
no_dose_reg = (\lambda d : \text{dose\_reg} @ \text{blank})
no_dose = (\text{p\_dose} \rightarrow \text{blank}, \text{p\_time} \rightarrow \text{blank})

---NoFieldF---

Field

prescribed = no_prescrip
accumulated = no_counter
no_dose_subseteq computed
overridden = {}

NoFieldS =^= [\text{Session} | \text{field} = \text{no\_field}]
NoField =^= NoFieldF and NoFieldS
FieldSession =^= PrescribedField or NoField

---'Initialization'---

----InitField----
NoFieldF

computed = calibrated =
no_dose_reg + \text{cal\_factor}
---SelectPatient---

Delta Field

NoField'
'computed' = computed + no_dose
'calibrated' = calibrated

---Select_Fiel---

Delta Field

---NewField---

Delta Field

---SelectExptField---

NewFieldF

mode = experiment
'computed' = computed
'calibrated' = calibrated

DOSE == VALUE; RATE == VALUE; FACTOR == VALUE; TIME == VALUE

t_backup : (DOSE x RATE x FACTOR )
    -- TIME

forall d : valid dose,
r : valid d_rate ; f : valid t_fac @
(d,r,f) in dom_t_backup and
t_backup(d,r,f) in valid p_time

---DoseTime---

Delta Field

(let t == t_backup(computed'
p_dose, computed'
d_rate,
computed'
t_fac) @
calibrated' = calibrated +
{p_time -> t})
'computed' p_time =
calibrated' p_time
{p_dose, p_time} <+ computed' =
{p_dose, p_time} <+ computed

---NewTherapyField---

NewFieldF

DoseTime

mode = therapy
accumulated' = counters field'

---SelectTherapyField---

NewTherapyField

computed' p_dose =
prescribed dose - accumulated dose
overridden' = {}

SelectSimpleFieldF =^ SelectExptFieldF or
SelectTherapyFieldF
```plaintext
selectComplexField?  
NewTherapyField  
dose? : VALUE

computed' p_dose = dose?  
  (let ovr = lambda c : counter |  
  accumulated' c >= prescribed' c @  
  accumulated c @  
  overridden' =  
  if dose? = prescribed' dose then  
  ovr else ovr union  
  {dose -> dose?})

Edit_setting

EditF
  Delta Field  
  item? : ITEM  
  value? : VALUE

  accumulated' = accumulated  
  calibrated' = calibrated

EditPresetF
  EditF
  item? in preset
  prescribed' = prescribed +  
  (item? -> value?)  
  overridden' =  
  (item?) <+ overridden  
  computed' = computed

EditCalF
  EditF
  item? in dose_reg\  
  {p_dose, p_time}  
  computed' = computed +  
  {item? -> value?}

  prescribed' = prescribed  
  overridden' = overridden

EditDoseF
  EditF
  DoseTime
  item? = p_dose  
  computed' p_dose = value?  
  overridden' = overridden +  
  {dose -> value?}

  prescribed' = prescribed

EditTimeF
  EditF
  item? = p_time  
  prescribed' = prescribed  
  computed' = computed +  
  {p_time -> value?}

  prescribed' = prescribed  
  overridden' = overridden

EditSettingF =~= EditCalF or EditPresetF or  
  EditDoseF or EditTimeF
```

---

117
Override

---OverrideF---
Delta Field
item? : ITEM

- prescribed' = prescribed
- accumulated' = accumulated
- computed' = computed
- calibrated' = calibrated

Override_Setting

---OverrideF---
item? in prescr
overridden' = if item notin dom
overridden then overridden +
  (item? -> measured item?)
else {item?} <+ overridden

---OverrideDose---

OverF

item? in {p_dose, p_time}
overridden' =
if dose notin dom overridden
then overridden +
  (dose -> accumulated dose)
else {dose} <+ overridden

OverrideF =~= Override_Setting or OverrideDose

Store_Field

zero_counter == (lambda c : counter @ 0)

---StoreFieldF---
Delta FieldSession

computed' = computed + no_dose
prescribed' = prescribed + (prescrip
  <| measured) + no_counter + {n_frac
  -> 1}
accumulated' = zero_counter
overridden' = {}

---ExptMode---

Delta Field

NoFieldF
computed' = computed + no_dose

calibrated' = calibrated

Calibration_factors

PRESSURE == VALUE; TEMPRATURE == VALUE

pt_formula : (PRESSURE x TEMPRATURE) -> FACTOR

forall p : valid press, t : valid temp @
  (p, t) in dom pt_formula and
  pt_formula(p, t) in
  valid pt_factor

/* */
PT_Auto = Field (computed pt_mode = automatic)

--- PTIntlk ---
Interlk
Field

interlock pt_intlk = if (PT_Auto and PT_Drift) or 
PT_Invalid or T_Invalid then set else clear.

--- TherapySumIntlk ---
Intlk

therapy_sum = if set in interlock (|sw_intlk|). 
then set else clear

************************************+***** j
* * * *setting_the_software_interlocks_and_flags ***********************
scanned == { gc_cc, fw_cc, fw_fault, lc_cc, 
pt_intlk, dmc_cal}

--- ScanIntlk ---
Xi Session
Xi Field
Delta Intlk

Status'
SS_Status'
FilterWedgeIntlk'
DoseCalIntlk'
TherapySumIntlk'
( run = setup and PTIntlk' ) or 
(run = running and 
interlock' pt_intlk = interlock pt_intlk)

scanned <+ interlock' = scanned <+ interlock|

******************************************************************************************************************

--- InitIntlk ---
Intlk

run = setup
therapy_sum = set
ran control = ( local )
ran drive = ( enabled )
ran inconsistent = ( set )
ran status = ran ss_status = ( not_ready )
interlock = (lambda i : INTERLOCK @ set) +
(lambda i:( setup_timeout, start_timeout, update) @ clear )

******************************************************************************************************************

****** User_Interface **************

--- Event ---
Delta Console
input? : INPUT

INPUT ::= filter_wedge | leaf_collim | dose_intlk | gantry_psa |
dose_cal | startup | help | messages select_patient | select_field |
field_summary | login | edit_setting | edit_dose_reg | log_message |
store_field | override_cmd | cancel_run | password | auto_setup |
OP : INPUT

- OP = { filter_wedge, leaf_collim, dose_intlk, gantry_psa, dose_cal, startup, help, messages, select_patient, select_field, field_summary, login, edit_setting, edit_dose_reg, log_message, store_field, override_cmd, cancel_run, password, auto_setup, expt_mode, cancel, refresh, shutdown, select}

DISPLAY P : OP

- DISPLAY = { filter_wedge, leaf_collim, dose_intlk, gantry_psa, dose_cal, startup, help, messages, select_patient, select_field, field_summary, login}

*****Console_state*****************************

INTERACTION ::= available | dialog | menu | confirm
nmax : N

- SELECTION == {i : N | i <= nmax}

[STRING]

- empty : STRING

KEYSWITCH ::= locked | unlocked

- RUN ::= setup | running

-----Console-------------------------

keyswitch : KEYSWITCH
run : RUN
display : DISPLAY
op : OP
interaction : INTERACTION
item : ITEM
nlist : P NAME
list_item : NAME
menu_item : SELECTION
buffer : STRING

-----InitConsole--------------------------

----Console-------------------------

op = login
display = login
interaction = dialog
buffer = empty

*****************************

*****Elements_of_User_Interaction*********

[ CAPTION, MESSAGE ]

- alert : CAPTION

-----Ignore---------------------

---Event---------------------

- Xi Console
caption! : CAPTION

---
caption! : alert

Unlocked =^= { Console | keyswitch = unlocked }

EventUnlocked =^= Event and Unlocked
Select $\Rightarrow$ EventUnlocked $\Rightarrow$ input? = select |

*****Available***************************************

Available $\Rightarrow$ [ Console | interaction = available ]

---------Op-------------------------------
    EventUnlocked
    |
    Available
    input? in OP

Setup $\Rightarrow$ [ Available | run = setup ]

Running $\Rightarrow$ [ Available | run = running ]

---------SelectDisplay-----------------------------
    Op
    |
    input? in DISPLAY
display' = input?
op' = display'
Available'

Engaged $\Rightarrow$ [ Console |
interaction /= available ]

---------Done-------------------------------
    EventUnlocked
    |
    Engaged
    op' = display
display' = display
Available'

Cancel $\Rightarrow$ [ Done | input? = cancel ]

******Lists***************************************

list : P DISPLAY
default_name : P1 NAME $\rightarrow$ NAME

---------forall list : P1NAME @ default_name list in list

List $\Rightarrow$ [ Available | display in list and nlist /= {} and list_item in nlist ]

---------SelectList-----------------------------
    SelectDisplay
    |
    input? in list
    ((nlist = {} and list_item' = no_name) or (List' and list_item' = default_name nlist'))

v_arrow == { up_arrow, down_arrow }

aname : (v_arrow x NAME X P1 NAME) $\rightarrow$ NAME

---------forall a : v_arrow; n : NAME; list : P1NAME @ aname (a, n, list) in list

Continue $\Rightarrow$ [ Delta Console | interaction' = interaction and op' = op and display' = display ]

---------GetListArrow-----------------------------
    EventUnlocked
    Delta List
    |
    input? in v_arrow
    list_item' = aname( input?,
selected_msg : NAME -> MESSAGE

---SelectName---
Select
name! : NAME
message! : MESSAGE

List
name! = list_item
message! = selected_msg name!

---Tables---

| table : P DISPLAY |
| default_item : table --> ITEM |
| table_items : table --&gt; PI ITEM |

forall d : table @ default_item d in table_items d

Table =^= [ Available | display in table and item in table_items display]

---SelectTable---
SelectDisplay

input? in table
item' = default_item display'
Table'

arrow == { right_arrow, left_arrow} union v_arrow

asetting : ( arrow x ITEM x table --> ITEM

forall a : arrow ;s : ITEM ; d : table @ asetting(a, s, d) in table_items d

---GetSettingArrow---
EventUnlocked
Delta Table

input? in arrow
item' = asetting( input?, item, display )
Continue

setting_table, dose_reg_table : P table
forall d : setting_table @ table_items d subseteq setting
forall d : dose_reg_table @ table_items d subseteq dose_reg

---SelectItem---
Select

Setup
Table
item' = item
( op' = edit_dose_reg and display in dose_reg_table or op = edit_setting and item in setting

---Editing---
Console
automatic, manual : VALUE

-----ScanPT------

Delta Field

calibrated' pt_factor = pt_formula (calibrated press, calibrated temp)
computed' pt_factor = pt_formula (calibrated press, computed temp)
(let pt_corr == if computed pt_mode = automatic then calibrated' pt_factor
else computed' pt_factor)
computed' calvolt1 = pt_corr * calibrated calvolt1 and computed' calvolt2 = pt_corr * calibrated calvolt2)

{ pt_factor } <+ calibrated' = {pt_factor}
{ calibrated' } <+ calibrated
{pt_factor, calvolt1, calvolt2 } <+ computed'
prescribed' = prescribed
accumulated' = accumulated
override' = overridden

Software_interlocks

***Intlk_state***

AUTO ::= local | auto
RUN ::= setup | running
ENABLE ::= enabled | disabled
READY ::= not_ready | ready | override

---Intlk---

run : RUN
therapy_sum : INTLK
interlock : INTERLICK --> INTLK
status : setting --> READY
ss_status : SUBSYS --> READY
drive : motion --> ENABLE
inconsistent : motion --> INTLK
control : motion --> AUTO

Relation_to_Session_and_Field

---IntlkFieldSession---

* dom overridden subseteq dom status
* dom status = if mode = therapy then prescr
  else preset

Setting_status_flags

|__| == ( lambda x : Z @ max(x, -x))
SETUP == setting --> VALUE
\[
\begin{align*}
\text{(let Match_)} &= \{ s : \text{setting, setup : SETUP} \\
&\ (s \text{ in selection and setup } s = \text{measured } s) \text{ or } (s \text{ in scale and} \\
&\ \text{setup } s - \text{measured } s \leq \text{tol } s) \} @ \\
\text{status} &= \\
&\ (\lambda s : \text{dom status} @ \text{not_ready}) + \\
&\ (\lambda s : \text{dom status} \ | \ \text{Match } (s, \ \text{prescribed}) @ \text{ready}) + \\
&\ (\lambda s : \text{dom status intersection counter} \ \\
&\ \text{accumulated } s < \text{prescribed } s @ \text{ready}) + \\
&\ (\lambda s : \text{dom overridden} \ | \ \text{Match}(s, \\
&\ \text{overridden}) @ \text{override}) + \\
&\ (\lambda s : \text{dom overridden intersection} \\
&\ \text{counter} @ \text{override}) + \\
&\ (\lambda s : \text{dom status} \ | \ \text{measured } s \notin \text{valid} \\
&\ s @ \text{not_ready}) )
\end{align*}
\]

*********Check_and_confirm_interlocks*************

-----SS_Status-----

| InterlkFieldSession |

(\let \text{ss_readiness} = (\lambda ss : \text{SUBSYS} @ \\
&\ \text{not_ready}) + \\
&\ (\lambda ss : \text{SUBSYS} \ | \ \text{not_ready} \ \text{notin status} \\
&\ (\{|\text{settings ss} \}| @ \text{override}) + \\
&\ (\lambda ss : \text{SUBSYS} \ | \ \text{ready} = \text{status} \\
&\ (\{|\text{settings ss}|\} @ \text{ready} @ \text{let (Check_Confirm}_{\text{target}} = \\
&\ \{|\text{ss:SUBSYS} \ | \ \text{ss_readiness} \\
&\ ss = \text{not_ready or enabled in drive} \\
&\ (\{|\text{settings ss}|\}) @ \text{or set in inconsistent} (\{|\text{settings ss}|\}) @ \text{interlock} \ \\
&\ \text{fw_cc} = \text{if CheckConfirm} (\text{filters}) \\
&\ \text{then set else clear and} \\
&\ \text{interlock lc_cc} = \text{if CheckConfirm} (\text{collimator}) \\
&\ \text{then set else clear and} \\
&\ \text{interlock gc_cc} = \text{if CheckConfirm} \\
&\ (\text{gantry_couch}) \text{ then set else clear and} \\
&\ \text{ss_status} = (\lambda ss : \text{SUBSYS} @ \text{if set in} \\
&\ \text{interlock} \\
&\ (\{|\text{interlocks ss}|\} ) \text{ then not_ready else} \\
&\ \text{ss_readiness ss)))

*********Other_software_interlocks*************

-----FilterWedgeInterlk-----

\text{Intlk} \\
\text{Field}

(\let \text{ms} = \{|\text{m : settings filters} | \\
&\ \text{drive m = disabled and measured m \notin valid m} \} @ \text{interlock} \text{fw_fault} = \text{if \text{ms} \neq 0} \text{ then set} \\
&\ \text{else clear}

-----DoseCalInterlk-----

\text{Intlk} \\
\text{Field}

(\let \text{cs} = \{|\text{c : cal_const} | \ \text{calibrated c \notin valid c} \} @ \text{interlock} \text{dmc_cal} = \text{if \text{cs} \neq 0} \text{ then set} \\
&\ \text{else clear}

\text{P_Invalid} \ominus \{\text{Field} | \ \text{computed press \notin valid press}\} \\
\text{T_Invalid} \ominus \{\text{Field} | \ \text{computed temp \notin valid temp}\} \\
\text{PT_Drift} \ominus \{\text{Field} | \ \text{computed pt_factor - calibrated pt_factor} | > \text{tol pt_factor} |}
interaction in \{ \text{dialog, menu} \}
(op = \text{edit_dose_reg} \text{and item in dose_reg}
\text{or} \ op = \text{edit_setting}
\text{and item in setting})

**********Confirm**********

Confirm \(=^\ast\) \{ \text{Console \mid interaction = confirm} \}

\text{ocaption : OP} \rightarrow \text{CAPTION}

---\text{ConfirmOp}---

\text{Op}
\text{caption!, query! : CAPTION}

\text{caption! = ocaption op'}
\text{display' = display}
\text{Confirm'}

AcceptConfirm \(=^\ast\) \text{Confirm and Select and Done}

\text{default_selection : SELECTION}

Menu \(=^\ast\) \{ \text{Editing \mid interaction = menu} \}

---\text{MenuOp}---

\text{Op}
\text{caption! : CAPTION}
\text{menu! : iseq\text{CAPTION}}

\text{menu_item' = default_selection}
\text{display' = display}
\text{Menu'}

\text{setting_info_name : ITEM} \rightarrow \text{CAPTION}
\text{setting_value : selection} \rightarrow \text{iseq \text{CAPTION}}

---

for all \text{s : selection} @
\text{dom( setting_value s )} = \text{valid s}

---\text{MenuEdit}---

\text{MenuOp}
\text{SelectItem}

\text{item in selection}
\text{caption! = setting_info_name item;}
\text{menu! = setting_value item}

\text{amenu : (v\_arrow x SELECTION x selection)} \rightarrow \text{SELECTION}

---

for all \text{s : selection} @
(\text{let n == #\text{valid s}}) @
\text{forall a : v\_arrow, i : SELECTION}
\text{default_selection <= n and amenu(a, i, s)) <= n)

---\text{GetMenuArrow}---

EventUnlocked
\text{Delta Menu}

\text{input? in v\_arrow}
\text{menu_item' = amenu( input?, menu_item, item)}

Continue

AcceptMenu \(=^\ast\) \text{Menu and Select and Done}

---\text{MenuSettingC}---

AcceptMenu
\text{item! : ITEM}
\text{value! : VALUE}
\[
\text{item!} = \text{item} \\
\text{value!} = \text{menu_item}
\]

*****Dialog*****************************

\[\text{Dialog} \leftarrow \{ \text{Console} \mid \text{interaction} = \text{dialog} \]\n
---DialogOp---

\[\text{Op} \]
\[
\text{caption!}, \text{prompt!} : \text{CAPTION}
\]
\[
\text{caption!} = \text{ocaption op'}
\]
\[
\text{display'} = \text{display Dialog'}
\]
---STRING, CHAR : P INPUT---

\[\text{modify} : (\text{STRING} \times \text{CHAR} \rightarrow \text{STRING}\]

---GetChar---

\[\text{EventUnlocked} \]
\[\text{Delta Dialog} \]
\[
\text{input? in CHAR}
\]
\[
\text{buffer'} = \text{modify(buffer, input?)}
\]
\[
\text{Continue}
\]
---Accept---

\[\text{Done} \]
---Dialog---

\[
\text{input? in terminator}
\]
---Reprompt---

\[\text{EventUnlocked} \]
\[\text{Delta Dialog} \]
\[
\text{input? in terminator}
\]
\[
\text{buffer'} = \text{empty}
\]
\[
\text{Continue}
\]

MIN == VALUE ; MAX == VALUE

---setting_info : ITEM \times MIN \times MAX \rightarrow \text{CAPTION}---

---DialogEdit---

\[\text{DialogOp} \]
\[\text{SelectItem} \]
---item notin selection---

\[
\text{prompt!} = (\text{let } v = \text{valid item} \in \text{setting_info}(\text{item, min } v, \text{max } v))
\]
---sval : STRING \rightarrow VALUE---

---EditSettingC---

\[\text{Accept} \]
\[\text{item!} : \text{ITEM} \]
\[\text{value!} : \text{VALUE} \]

---Editing---

\[
\text{let } v = \text{sval buffer} \in \text{valid item and value!} = v
\]

---InvalidSetting ---

\[\text{InvalidSetting} \leftarrow \{ \text{Reprompt} \mid \text{Editing and} \]
\[\text{sval buffer notin valid item} \}

---EditOrInvalidSetting---

\[\text{EditOrInvalidSetting} = \text{EditSettingC} \lor \text{InvalidSetting} \]
**Therapy_console_operations**

--- **Relation_to_Session_state** ---

--- ConsoleSession ---

Console
Session

display = select_patient implies
nlist = names

display = select_field implies nlist =
dom fields

--- **Op_operations** ---

SimpleOp = Op and Continue

--- ExptModeC ---

SimpleOp
Xi Session
Delta ConsoleSession

Setup
input? = expt_mode
operator in physicists

auto_setup_display == { field_summary,
filter_wedge, leaf_collim, dose_intlk}

--- AutoSetupC ---

SimpleOp
Xi ConsoleSession
subsystem! : auto_setup_display

Setup
field /= no_field
display in auto_setup_display
input? = auto_setup
subsystem! = display

--- **Select_Display_Operations** ---

simple_display == { field_summary, help }

--- **Select_List_Operations** ---

list = { select_patient, select_field }

--- SelectPatientList ---

SelectList
Xi Session
Delta ConsoleSession

input? = select_patient
nlist' = names

--- SelectFieldList ---

SelectList
Xi Session
Delta ConsoleSession

patient /= no_patient
input? = select_field
nlist' = dom fields

--- **SelectTableOperations** ---

table = { gantry_psa, filter_wedge, leaf_collim,
**table_items** = { gantry_psa -> { gantry, collim, turnt},
    { filter_wedge -> { filter, wedge, w_rot}}
leaf_collim -> leaves,
dose_intlk -> { p_dose, p_time},
dose_cal -> { pt_mode, press, temp, d_rate, t_fac}

setting_table = { gantry_psa, filter_wedge, leaf_collim}
dose_reg_table = { dose_intlk, dose_cal }

**********ConfirmOpOperations**************

<table>
<thead>
<tr>
<th>cancel_run_query : CAPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----SelectCancelRun------</td>
</tr>
<tr>
<td>ConfirmOp</td>
</tr>
<tr>
<td>Running</td>
</tr>
<tr>
<td>input? = cancel_run</td>
</tr>
<tr>
<td>op' = input?</td>
</tr>
<tr>
<td>query! = cancel_run_query</td>
</tr>
</tbody>
</table>

**********DialogOp_Operations**************

<table>
<thead>
<tr>
<th>type_message_prompt, store_field_prompt:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----TypeMessage-----</td>
</tr>
<tr>
<td>DialogOp</td>
</tr>
<tr>
<td>input? = log_field</td>
</tr>
<tr>
<td>op' = input?</td>
</tr>
<tr>
<td>prompt! = type_message_prompt</td>
</tr>
</tbody>
</table>

| -----EditField-----                    |
| DialogOp                               |
| Setup                                  |
| input? = store_field                   |
| op' = input?                           |
| prompt! = store_field_prompt           |

**********Setup_operations**************

| -----SelectPatientC-----               |
| SelectName                             |
| Setup                                  |
| display = select_patient               |
| Continue                               |

| -----NewFieldC-----                    |
| SelectName                             |
| ConsoleSession                        |
| Setup                                  |
| display = select_field                 |

| -----SelectSimpleFieldC-----           |
| NewFieldC                             |
| mode = experiment                      |
| or ( counters name! dose = 0 and not exceeded( fields name!, counters name!)) |
| Continue                               |
DoseDialogOp

NewFieldC
DialogOp
Delta Console

'op' = select_field
new_field' = name:
mode = therapy

Console

ConsoleSession
new_field : FIELD

new_field in dom fields

sprintf : VALUE -> STRING
delivered_prompt : NAME x VALUE x VALUE -> CAPTION

SelectDeliveredField

DoseDialogOp

counters new_field' > 0
not exceeded( fields new_field',
counters new_field')

(let d == fields new_field' dose;
c == counters new_field' dose @
(let default_dose == d - c @ buffer' =
and prompt! = delivered_prompt
( new_field', d, c, default_dose))

exceeded_prompt : NAME x ACCUMULATION x ACCUMULATION -> CAPTION

SelectExceededField

DoseDialogOp

exceeded( fields new_field',
counters new_field')
prompt! = exceeded_prompt (new_field',
fields new_field', counters new_field')
buffer' = empty

SelectFieldC *= SelectSimpleFieldC or
SelectDeliveredField or SelectExceededField

SelectFieldOp

Console

op = select_field

override_table == { filter_wedge, leaf_collim,
gantry_psa, dose_intlk }

override_query : CAPTION -> CAPTION

SelectOverride

ConfirmOp
SelectItem
Xi Session

field /= no_field
display in override_table
input? = override_cmd
op' = input
query! = override_query

( setting_info_name item)

cal_table == { dose_cal }

129
CalTable = {ConsoleSession | mode = experiment and display in cal_table}

SelectCalMenu =^= CalTable and MenuEdit
SelectCalDialog =^= CalTable and DialogEdit
SelectSettingMenu =^= SettingTable and MenuEdit
SelectSettingDialog =^= SettingTable and DialogEdit

Cancel_Operations

Loggedln =^= [Console | op notin {login, password}]

CancelOp =^= Loggedln and Cancel

Accept_Confirm_Operations

OverrideC

AcceptConfirm
item! : ITEM

op = override_cmd
item! = item

CancelRunC =^= [AcceptConfirm | op = cancel_run]

Accept_Operations

log_msg : STRING --> MESSAGE

WriteMessageC

Accept
message! : MESSAGE

op = log_message
message! = log_msg buffer

sname : STRING --> NAME
store_msg : NAME --> MESSAGE

StoreFieldC

Accept
field! : NAME
message! : MESSAGE

op = store_field
field! = sname buffer
message! = store_msg field!

SelectComplexFieldC

Delta Console

Accept
field! : FIELD
dose! : VALUE

SelectFieldOp
field! = new_field
( let d == sval buffer @ d in valid dose and dose! = d )
InvalidDose = Reprompt: Delta Console1, SelectFieldOp and sval buffer notin valid dose ]
ComplexOrInvalidField = SelectComplexFieldC or InvalidDose

Logout_and_Login
Logoff
op = login
input? = login
op' = display'
buffer' = empty
message! = lo_msg operator
Dialog'

Console2
username : STRING

EnterUsername
Delta Console2
Delta Dialog
EventUnlocked
op = login
input? in terminator
username' = buffer
buffer' = empty
op' = password
display' = display

USERNAME == STRING ; PASSWORD == STRING
soper: (USERNAME x PASSWORD) -> OPERATOR

LoginC
DeltaConsole2
EventUnlocked
operator! : OPERATOR
message! : MESSAGE

op = password
soper( username, buffer) in operators
input? in terminator
display' = help
op' = display
operator! = soper( username, buffer)
message! = o_message operator!
Available'

Unauthorized
Delta Console2
Reprompt
op = password
soper( username, buffer) notin operators
username' = username

LoginOrUnauthorized = LoginC or Unauthorized

---CancelUsername---
Even tuned
login, password
input? = cancel
op' = login
buffer' = empty
username' = empty
display' = display

Other_Operations

Refresh ^= [ Event | input? = refresh ]
Shutdown ^= [ Event | input? = shutdown ]

Implementation

Physicist ^= [ ConsoleSession | operator in physicists ]
PatientList ^= [ Console | display = select_patient and nlist /= {} ]
PatientSelected ^= [ ConsoleSession | patient /= no_patient ]
FieldList ^= [ Console | display = select_field and nlist /= {} ]
FieldSelected ^= [ ConsoleSession | field /= no_field ]
AutoSetupDisplay ^= [ Console | display in auto_setup_display ]
OverrideTable ^= [ Console | display in override_table ]
MenuItem ^= [ Console | item in selection]
DialogItem ^= [ Console | item notin selection]
LoggedOut ^= [ Console | op in { login, password }]
OverrideOp ^= [ Console | op = override_cmd ]
CancelRunOp ^= [ Console | op = cancel_run ]
LogMessageOp ^= [ Console | op = log_message ]
StoreFieldOp ^= [ Console | op = store_field ]
UsernameOp ^= [ Console | op = login ]
PasswordOp ^= [ Console | op = password ]
Input ^= [ Input | input? : INPUT ]
DisplayKey ^= [ Input | input? in simple_display ]
PatientKey ^= [ Input | input? in select_patient ]
FieldKey ^= [ Input | input? in select_patient ]
TableKey ^= [ Input | input? in table ]
MessageKey ^= [ Input | input? = log_message ]
VArrowKey ^= [ Input | input? in v_arrow ]
SelectKey ^= [ Input | input? = select ]
ArrowKey ^= [ Key ^= [ Input | input? Input | input? in arrow ] }
ExptModeKey ^= \{ Input | input? = expt_mode \}
AutoSetupKey ^= \{ Input | input? = auto_setup \}
StoreFieldKey ^= \{ Input | input? = store_field \}
LoginKey ^= \{ Input | input? = login \}
OverrideKey ^= \{ Input | input? = override_cmd \}
CancelRunKey ^= \{ Input | input? = cancel_run \}
CancelKey ^= \{ Input | input? = cancel \}
CharKey ^= \{ Input | input? \in CHAR \}
TerminatorKey ^= \{ Input | input? \in terminator \}
RefreshKey ^= \{ Input | input? = refresh \}
ShutdownKey ^= \{ Input | input? = shutdown \}

---------------------------------------------------------------

********Combining_the_subsystems*******************************
ExptMode ^= ExptModeC and ExptModeS and ExptModeF

----Override---------------------
| OverrideC
| OverrideF
| item! = item?
| Override ^= OverrideC >> OverrideF

Login ^= LoginC >> LoginS

EditSetting ^= EditSettingC >> EditSettingF

StoreField ^= StoreFieldC >> ( StoreFieldS and StoreFieldF )

SelectPatient ^= SelectPatientC >> ( SelectPatientS and SelectPatientF )

SelectSimpleField ^= SelectSimpleFieldC >> ( SelectFieldS and SelectSimpleFieldF )

SelectComplexField ^= SelectComplexFieldC >> ( SelectFieldS and SelectComplexFieldF )

end specification