Stimulus Response Requirements Specification Notation: An Empirically Evaluated Requirements Specification Notation

by

KENDRA M. L. COOPER

M.A.Sc. The University of British Columbia, 1995

B.A.Sc. The University of British Columbia, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

THE UNIVERSITY OF BRITISH COLUMBIA

March 2001

© Kendra M.L. Cooper, 2001
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.

Department of Electrical and Computer Engineering

The University of British Columbia
Vancouver, Canada

Date March 30, 2001
Abstract

This research has focussed on the development of a formal requirements specification notation that is suitable for the specification of large scale systems with complex data and logical requirements. Examples of such systems include air traffic control systems and automated on-line library systems. The motivation for defining the stimulus response requirements specification (SRRS) notation is to provide a notation that is readable (like natural language) and yet at the same time is amenable to automated tool support. Automated tool support such as parsers, typecheckers, analysis tools, and translators are extremely valuable in the effort to make the requirements specification phase of software development better, faster, and cheaper. The quality of the requirements specification may be improved as authors use the tool support to identify and remove defects in the specification before a peer review. The improvement in the quality of the specifications allows a peer inspection to take place in less time. The improvement in quality and the reduced inspection time means that the requirements specifications cost less to create. Additional quality, time, and cost improvements may be made by automating the generation of test specifications for the system level testing phase in the software development lifecycle.

The SRRS notation is evaluated in a controlled experiment to determine the costs and benefits of using the notation with its tool support in comparison to a similar, semi-formal notation. The experimental results show a significant reduction in defects detected in a peer review (81%) and a significant reduction in the amount of time to write, review, and correct a specification (39%). The costs include an increase in the training time: the subjects need two days of training, however, instead of one day.

The problem of defining a requirements specification notation that is suitable for describ-
ing systems with intricate data and logical conditions is complex. The decomposition of the problem led to a divide and conquer approach that solves the problem. The result is a set of two notations. The first notation (SRRS) is tailored for specifying requirements. The second notation is a core, or foundation, notation that is tailored to manage the data and logical condition complexity. The second notation is called the data specification (DSPEC) notation. As a core notation, DSPEC may be re-used with other notations that are well suited for other phases of the lifecycle. For example, a system level test specification notation could be defined that re-used the DSPEC notation.

The process of developing the formal SRRS notation is also presented in this work. The process begins with a notation that is currently being used at Raytheon Canada Systems Ltd. called Thread-Capability. The notation is evaluated and updated to create the semi-formal SRRS. The semi-formal notation is evaluated in industry in a case study and the notation is formalized by defining it syntax and semantics. A generalized process for formalizing an existing notation is included so that the work may be re-used.
# Table of Contents

Abstract .............................................................................................................. ii

List of Tables ........................................................................................................ x

List of Figures ....................................................................................................... xi

List of Acronyms .................................................................................................. xii

Acknowledgment ................................................................................................. xv

Chapter 1  Introduction ......................................................................................... 1

1.1 Requirements Specification Document ...................................................... 1

1.2 History of the Research Project ................................................................. 3

1.3 Contributions to Knowledge .................................................................. 4

1.4 Intellectual Challenges .......................................................................... 7

1.4.1 Scope of Research Project ............................................................... 7

1.4.2 Breadth of Related Research Topics ............................................. 8

1.4.3 SRRS .............................................................................................. 9

1.4.4 DSPEC .......................................................................................... 11

1.4.5 Evaluation of SRRS ................................................................... 12

1.4.6 Process to Formalize a Notation ................................................... 12

1.5 Organization of Dissertation ................................................................ 12

Chapter 2  Related Work .................................................................................... 14

2.1 Introduction .............................................................................................. 14

2.2 Natural Language Techniques ............................................................... 14

2.2.1 Use Cases ..................................................................................... 16

2.3 Formal Methods ..................................................................................... 18
Chapter 6 Tool Support for SRRS and DSPEC .................................................. 124
6.1 Introduction ......................................................................................... 124
6.2 Software Requirements ................................................................. 124
6.3 Architecture Model ........................................................................ 125
   6.3.1 Structured Analysis Diagrams .................................................... 126
   6.3.2 Primitive Specifications ............................................................... 130
   6.3.3 Data Dictionary ......................................................................... 138
6.4 Implementation Summary .............................................................. 143
6.5 Future Enhancements ....................................................................... 143
6.6 Conclusions ....................................................................................... 144

Chapter 7 Experimental Evaluation of SRRS ........................................ 146
7.1 Introduction ....................................................................................... 146
7.2 Experimental Design ........................................................................ 146
   7.2.1 Problem Statement .................................................................. 146
   7.2.2 Experimental Design Overview ................................................. 149
7.3 Experimental Results ........................................................................ 153
   7.3.1 Defect Rates ............................................................................ 153
   7.3.2 Training Time .......................................................................... 155
List of Tables

Table 2.1 BNF Notation...........................................................................................................30
Table 2.2 ERD Symbols .........................................................................................................33
Table 2.3 Requirements Specification Notations.................................................................41
Table 3.1 Translation of a Predicate Declaration in SRRS into a Constant Declaration in S50
Table 4.1 Translating the SRRS Context to the DSPEC context........................................73
Table 4.2 Translation of SRRS to S.....................................................................................74
Table 5.1 Cardinality and Participation Constraints........................................................102
Table 5.2 Example of Parent Rule Evaluations ................................................................107
Table 5.3 Data Expression: Example of Basic Data Entity Reference ...............................112
Table 5.4 Data Expression: Examples of Association References ....................................113
Table 5.5 Logical Condition Examples .............................................................................115
Table 5.6 Quantified Expressions ......................................................................................116
Table 5.7 Summary of Translating DSPEC Components Into S........................................117
Table 5.8 Example Translation of Declarations in DSPEC into S.....................................119
Table 5.9 Example Translation of Data Expressions in DSPEC into S ...............................120
Table 6.1 Implementation Effort Summary ........................................................................142
Table 7.1 Summary of Defects Recorded ...........................................................................153
Table 7.2 Summary of Training Time ................................................................................155
Table 7.3 Summary of Writing and Reviewing Time............................................................157
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>A Taxonomy of Requirements Notations and Selected Examples</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Object Model Symbols</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>A Five Step Process to Formalize the Thread-Capability Notation</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>A Five Step Process to Formalize a Notation</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Evolution and Influences on the SRRS Notation</td>
<td>56</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Structured Stimulus Response Specification Unit Structure</td>
<td>64</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Simple Example of DSPEC Building Paths</td>
<td>97</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Data Specification Structure</td>
<td>99</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Directed Graph of Data Specification Example</td>
<td>107</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Re-use of DSPEC as a Foundation Notation</td>
<td>122</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Context Diagram</td>
<td>125</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Diagram 1 SRRS System</td>
<td>126</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Diagram 1.1 Parse Specification</td>
<td>127</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Diagram 1.2 Validate Specification</td>
<td>128</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Diagram 1.3 Translate Specification</td>
<td>129</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>State Transition Diagram for CSPEC 1.1.1</td>
<td>130</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>State Transition Diagram for CSPEC 1.3.1</td>
<td>131</td>
</tr>
<tr>
<td>Figure 6.8</td>
<td>State Transition Diagram for CSPEC 1.2.1</td>
<td>132</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>A Six Step Process to Develop a Specification Unit</td>
<td>150</td>
</tr>
<tr>
<td>Figure 8.1</td>
<td>Evolution and Influences on the SRRS notation</td>
<td>165</td>
</tr>
<tr>
<td>Figure 8.2</td>
<td>Re-use of DSPEC as a Foundation Notation</td>
<td>171</td>
</tr>
</tbody>
</table>
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>American National Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ACE</td>
<td>Attempto Controlled Language</td>
</tr>
<tr>
<td>ANLT</td>
<td>Alvey Natural Language Toolkit</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur form</td>
</tr>
<tr>
<td>CAATS</td>
<td>Canadian Automated Air Traffic Control System</td>
</tr>
<tr>
<td>CHI</td>
<td>Computer Human Interface</td>
</tr>
<tr>
<td>CTL</td>
<td>Computation Tree Logic</td>
</tr>
<tr>
<td>DSPEC</td>
<td>Data Specification Notation</td>
</tr>
<tr>
<td>ERD</td>
<td>Entity Relationship Diagram</td>
</tr>
<tr>
<td>flex-fix</td>
<td>Flexible-fix Notation</td>
</tr>
<tr>
<td>Ha</td>
<td>alternate hypothesis</td>
</tr>
<tr>
<td>Ho</td>
<td>null hypothesis</td>
</tr>
<tr>
<td>HOL</td>
<td>Higher Order Logic</td>
</tr>
<tr>
<td>IOLS</td>
<td>Integrated On-line Library System</td>
</tr>
<tr>
<td>lex</td>
<td>Lexical analyser</td>
</tr>
<tr>
<td>Term</td>
<td>Abbreviation</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
</tr>
<tr>
<td>KSLOC</td>
<td></td>
</tr>
<tr>
<td>MAATS</td>
<td></td>
</tr>
<tr>
<td>NLP</td>
<td></td>
</tr>
<tr>
<td>Promela</td>
<td></td>
</tr>
<tr>
<td>ROID</td>
<td></td>
</tr>
<tr>
<td>RSML</td>
<td></td>
</tr>
<tr>
<td>SA</td>
<td></td>
</tr>
<tr>
<td>SADT</td>
<td></td>
</tr>
<tr>
<td>SCR</td>
<td></td>
</tr>
<tr>
<td>SDLC</td>
<td></td>
</tr>
<tr>
<td>SF SRRS</td>
<td></td>
</tr>
<tr>
<td>SFTA</td>
<td></td>
</tr>
<tr>
<td>SMV</td>
<td></td>
</tr>
<tr>
<td>SRRS</td>
<td></td>
</tr>
<tr>
<td>SRS</td>
<td></td>
</tr>
<tr>
<td>UML</td>
<td></td>
</tr>
</tbody>
</table>
VDM-SL  Vienna Development Method Specification Language

YACC  Yet Another Compiler Compiler
Acknowledgment

I would like to thank my supervisor Dr. Ito for his assistance and guidance in this research project.

I would like to acknowledge the generous funding provided by NSERC (Grant # 58-9054 and PGS B), B.C. ASI, UBC, FormalWare, and the Killam Foundation.
Chapter 1  Introduction

1.1  Requirements Specification Document

The requirements specification document is a key document in the software development lifecycle. It is used as the contract between the developer and the customer and as a technical working document by the architecture, system level test, and user documentation teams. Because of its early place in the lifecycle, mistakes that are made in the requirements specification documents can be the most costly to fix. Rework in multiple documents and/or source code files increases the cost of correcting a mistake for each phase it is carried through. An error introduced in the requirements specification phase that is not detected until the system has been deployed is estimated to cost 200 times what the correction costs if it is found and corrected in the specification phase [Dav93]. Processes, notations, and tool support for detecting and correcting as many defects as possible when writing the requirements are an essential part of reducing the cost of development. The better the requirements specification, the less rework that must be done and, as a result, the lower the development costs.

The purpose of the processes, notations, and tool support available for use in requirements engineering is to assist the authors in writing high quality requirements specifications. Characteristics of good quality requirement specifications are described in [Dav93, IEE93]. High quality requirements specifications are (among other things) understandable or readable, unambiguous, concise, precise, and verifiable.

Requirements specification notations that are used to maximize the understandable characteristic of a specification are based on natural language. The premise here is that requirements specifications written in natural language are the easiest to understand [Won94]. An
unstructured natural language specification is a description of the requirements that is constrained only by technical writing rules and formatting standards [Chi93]. Lightly structured natural language approaches include use cases. Currently available commercial tool support for natural language approaches is limited to spelling checkers and grammar checking tools.

To improve the understandability and precision of requirements specifications, semi-formal specifications have been developed that place additional constraints on the structure of the specification and the vocabulary used in the natural language component of the notation. Examples of semi-formal requirements specification notations include structured analysis and pseudo-natural language notations including a template approach, Thread-Capability, Attempto Controlled English (ACE), and Newspeak. The tool support for structured analysis techniques include computer aided software engineering (CASE) tools that support validation checks on the data and control flow diagrams in a specification. The template and the Thread-Capability notation have no tool support available. Tool support for ACE includes a parser and translator (to Prolog). Tool support for Newspeak includes a parser.

The formal notations emphasize the unambiguous, concise, and precise qualities for requirements specifications. Examples of mathematical, text based formal notations include Z and VDM-SL. The mathematician can use Z, for example, to introduce mathematical formulae in a specification that are abbreviations for lengthy English sentences [Spi88]. The tool support for Z and VDM-SL include typecheckers and theorem provers. Tabular specifications including the software cost reduction (SCR) approach offer a concise and precise technique for specifying complex relationships in an organized format. Tool support for tabular specifications include consistency checkers, model checkers, and simulators.
1.2 History of the Research Project

This research project has been part of the formalWARE project, a collaborative research project involving The University of British Columbia, The University of Victoria, Raytheon Systems Canada, Ltd., MacDonald Dettwiler, and the British Columbia Advanced Systems Institute. The formalWARE project focuses on the use of formal methods to address specific challenges in areas of system and software engineering such as requirements specification.

The requirements specification problem addressed in this research project has its origins in industrial projects that are under development by Raytheon Systems Canada Ltd., the Canadian Automated Air Traffic System (CAATS) and the Military Automated Air Traffic Control System (MAATS). These large scale projects both have complex functional requirements and complex data requirements. When completed, CAATS is expected to be the world’s most advanced automated air traffic management system.

The requirements specification notation used to capture the software requirements for the CAATS and MAATS projects is an in-house notation called “Thread-Capability”. A thread is similar in its purpose and scope to a concrete use case in the unified modelling language (UML). A thread describes a specific task the user needs to perform. It includes the specification of the normal case and the error cases. An example of a thread in a library system is that a borrower needs to be able to sign out a book. A capability is similar in its purpose and scope to an abstract use case in UML. A capability describes part of a task that is re-used in many threads. For example, updating the records for a user’s account may be used by a number of threads including paying a fine, returning a late book, and reporting a book as lost. The capability is used to describe part of a task in one place, rather than duplicating the description in multiple threads. This simplifies the maintenance of the requirements as modifications to this part of the description are
Chapter 1 Introduction

isolated in one capability.

The characteristics of the Thread-Capability notation that are its main strengths include the:

- external partitioning of the requirements
- blackbox style
- abstraction, or grouping, of the stimuli and responses
- readability of the notation
- highly structured format
- definition of terminology with a data dictionary

The Thread-Capability notation has a number of drawbacks that prevent its widespread use. These disadvantages include:

- lack of tool support to assist the authors in detecting defects
- difficulty in describing complex logical conditions
- lack of tool support to automate the development of system level (requirements based) testing
- lack of training material
- proprietary

1.3 Contributions to Knowledge

There are four primary contributions by this research. The first contribution of this work is to objectively analyze the costs and benefits of using the new SRRS notation and tool support. An experiment is designed, run, and analyzed to compare the costs (training time) and benefits (defect rates, development time) of using the formal version of the notation to its semi-formal
version. In order to run this experiment, tool support is developed for the SRRS notation. The motivation for this objective is to determine if this formal notation has the potential to be used in industry. The experimental evaluation of SRRS is an important contribution. In general, this work promotes the objective evaluation of new requirements specification techniques. In this specific work, the experimental evaluation demonstrates the costs and benefits of using SRRS in comparison to the semi-formal version of the notation. The results indicate the SRRS technique is very promising. The results of the experimental evaluation of the SRRS notation are also used to develop an initial set of estimation metrics for the training time and development of requirement specifications from a system level specification. Such estimation metrics are currently not available.

The second contribution of this research is to define a new process to develop the SRRS notation. The process has five steps and describes a systematic refinement of an existing notation. The first step is an evaluation of the existing notation for strengths and weaknesses. In the second step, the results are used to update the notation and correct some of the identified problems. When this is done, the updated notation is evaluated and error checks are identified in the third step. The fourth step is the formalization of the notation (defining the syntax and semantics) and defining the error checks needed in the notation. The resulting formal notation is objectively evaluated for its costs and benefits in the last step. This process is also generalized in this research to consider the general problem of formalizing an existing notation. The new, generalized process may be reused by other researchers who are investigating the formalization of an existing notation.

The third contribution of this research is to define and develop the DSPEC notation. DSPEC combines the flexibility of ERDs with the ability to check a data reference in a require-
ment automatically. In contrast, if a data item is referenced in a process specification in a structured analysis model, for example, the author must manually check the data dictionary or the ERD to determine if the data reference is unambiguous. DSPEC is also re-usable which allows the notation to be used in different phases of the software development lifecycle. For example, either a requirements specification or a test specification notation can use the underlying core language. This may be useful, for example, when test specifications are needed to evaluate third party software. The core notation is amenable to automated tool support for detecting defects in the data references.

The fourth significant contribution is the development of a formal version of a requirements specification notation called SRRS that is suitable for use in industry. The new, formal notation is based on the Thread-Capability notation that has been used at Raytheon Canada CAATS and MAATS projects [Hug93, Pai93]. The motivation for the fourth objective is make a notation that retains the strengths of the Thread-Capability notation and overcomes its weaknesses. The SRRS notation is understandable by customers and developers and is machine readable. The advantage of the machine readability is that the specification is amenable to automated tool support such as parsing, typechecking, and additional validation checks and the translation to other notations. Authors can use this tool support to remove defects in the requirements specifications earlier in the software development lifecycle (SDLC), making the development less expensive. To assist the author in detecting defects, two levels of checking are supported in SRRS. The first level supports the checks that are specific to the requirements specification notation. For example, if a response name is used in a requirement, then a check is made to determine if the name is declared. The second level of checking is used to detect defects in data relationships. For example, if a data type is referenced, then a check is made to determine
Chapter 1  Introduction

if the reference is unambiguous in the context. Defining the checks that are performed on the data and their relationships is a complex task. The engineering principle separation of concerns, or divide and conquer, is applied to the research problem. The solution is that the second level of checking on the data relationships is provided by a separate data specification language called DSPEC.

Due to the integration of the core language DSPEC, SRRS is suited for describing the requirements of systems that have complex data relationships. Examples of such systems include automated air traffic control and integrated on-line library systems.

1.4 Intellectual Challenges

There are significant intellectual challenges in this research. The scope of the research project is large, the breadth of related research topics is large, and each of the major components of the research problem provides challenges in developing a solution.

1.4.1 Scope of Research Project

As a whole, the problem of defining a new requirements specification notation that is readable and understandable (i.e., looks like natural language); flexible; compact; supports large scale, data intensive systems; amenable to tool support for the automated detection of defects; promotes a black-box style (input, processing, output) for writing the requirements; supports the automated translation of requirements into test specifications; and is cost effective to use is a large, complex problem.

Recognizing that the problem is too large to solve at one time, one of the initial challenges in the work is the question of how to divide the large, complex problem into a set of smaller, more
manageable problems. As a result of the investigation, the decision has been made to define two separate notations. The first is a requirement specification notation called SRRS. SRRS needs to be defined such that it has the characteristics listed earlier. Note that the SRRS notation does not manage the data model. Instead, an additional specification notation called DSPEC is defined to accomplish this. The relationship between these two notations is that the SRRS notation uses the DSPEC notation to manage the data and logical conditions in the requirements specification.

1.4.2 Breadth of Related Research Topics

The solution to this problem draws on a wide range of research topics areas including:

1. Natural language processing [Fuc99a, Osb96]. From natural language processing (NLP) research, the idea of using a constrained, or reduced, grammar in order to parse a body of text written in natural language is used. Current NLP research is focused on resolving ambiguities in parsing and using Prolog to develop tool support.

2. Template phrasing [Hen80]. Two ideas presented in the template phrasing introduced by Heninger are used. The first idea is to use template phrasing to constrain the grammar. The second idea is the removal of the restriction of using prefix, postfix, or infix forms. The template phrases presented in [Hen80] use a form that intersperses the parameter with the body of the predicate. This idea is termed “flex-fix” and is used in both the SRRS and the DSPEC notations.

3. Programming language definition [Ten81]. The foundations for programming languages are used in defining the specifications of the SRRS and DSPEC notations. Although not translated to source code, the input specification written in SRRS is scanned, parsed, typechecked, and has additional error checks performed on the input. The standard pipeline architecture for a
compiler is used in the architectural model of the SRRS notation. The Unix programming tools lex and yacc have traditionally been used to develop scanners and parsers for programming languages [Lev92]. The tools are applied in the development of the tools support for the SRRS and DSPEC notations.

4. Stimulus response style requirement specification notations [Jac92, Boo99, War85, Hug93]. The stimulus response notations partition the requirements specification from an external, or outside-in, perspective. The requirements are described in terms of inputs (stimuli), processing, and outputs (responses). This external partitioning is used in SRRS.

5. Data models [Dem79, Hat88, Hug93, Elm89, War85]. Data dictionaries and entity relationship models describe the data types and their relationships for a system. A data dictionary is restricted to describing the “is composed of” relationship. Entity relationship diagrams are used to graphically describe data and their relationships. A data context (data objects, data types, functions, predicates) is used in SRRS to support the textual description of data and their relationships. The data model is managed by the DSPEC notation.

6. Formal methods. The semantics of SRRS are defined using operational semantics. SRRS and DSPEC can be translated into a higher order logic called S. The advantage of this translation is the ability to automatically generate test specifications [Don98].

7. Design and analysis of experiments [Mon97, Por95]. An objective study of the SRRS notation is an important step in developing the notation for industrial use. The costs and benefits of using the notation need to be analyzed before applying the notation in industry.

1.4.3 SRRS

There are numerous challenges in solving the SRRS research problem. Retaining a readable, flexible English-like notation needs to be balanced with the ability to unambiguously
parse, analyze, and translate requirements specifications written in SRRS. To provide an English-like grammar, enough variety is needed to allow authors to write a requirements specification that reads smoothly but can still be unambiguously parsed, analyzed (validated), and translated. The flexibility in the grammar is achieved with a variety of requirements statements supported (stimulus response, response response, response qualification), optional placement of logical conditions, optional list formats, and an optional tabular format for logical conditions. The result of the flexibility in SRRS is that author has the choice of writing a requirement in a variety of different ways. Delineating logical conditions with curly braces is necessary for unambiguous parsing, analysis, and translation but does detract from the readability of the requirements.

The need to automate the validation of a requirements specification and to automatically generate test case specifications provide challenges. The errors must be defined and feasible to implement in tool support. In addition to parsing errors, SRRS has 162 validation checks defined and implemented in the tool support. To automatically generate test specifications, the interface to the test case generation tool (TCG) [Don98] needs to be investigated. Little documentation on the use of the TCG is available.

SRRS also needs to be a compact notation that allows the author to generalize, or abstract, the requirements that transform a set of inputs from different sources into a set of outputs that go to different destinations. In order to provide authors with this feature, the challenge is to define a mechanism in the notation that supports the generalization of inputs (stimuli) and outputs (responses). SRRS is defined with a set of thirteen rules that defines this mechanism.

In the implementation of the tool support, the SRRS notation requires approximately 15,000 lines of lex, yacc, and C source code to implement the grammar, operational semantics,
Chapter 1 Introduction

and the 157 validation checks defined in the notation.

1.4.4 DSPEC

The definition of the DSPEC notation has also provided intellectual challenges. The definition of an unambiguous data reference is needed. In DSPEC, graph theory is used to represent the data model. A data type is represented by a node in a graph, a relationship is represented by an arc between two data types. An unambiguous data reference (from one data type to another) occurs when there is exactly one path through the graph. The definition must consider the inclusion of cycles in the graph because these need to be detected and recorded in order to ensure the checks on the graph eventually stop.

DSPEC is flexible in that the notation allows the author to define and use domain specific data types and objects in addition to the objects and types that are built into the notation. A second characteristic that makes DSPEC flexible is that the notation allows the author to use data objects and data types interchangeably. For example, the author may use either a data object or a data type as an argument in a function application with DSPEC. In contrast a higher order logic, for example, only allows the author to use a data object as an argument. This flexibility is an important feature because when DSPEC is used by the SRRS notation, it allows authors to use either data objects or data types when describing the requirements for a system. The cost of this flexibility is the need to develop new validation rules that embody universal and existential instantiation in the typchecking.

In the implementation of the tool support, the DSPEC notation requires approximately 4,000 lines of lex, yacc, and C source code to implement the grammar, operational semantics, and the 31 validation checks defined in the notation.
1.4.5 Evaluation of SRRS

Another intellectual challenge in this research involves the question of how to objectively evaluate the SRRS notation. An empirical evaluation is selected as being the best choice; however, the literature in this area is quite limited. The challenges in designing this experiment centre on the trade-offs between the constraints of running an experiment (fixed budget) and the need to obtain meaningful results. Of the numerous options available, the experiment has been designed to test three key hypotheses: the defect detection rate, the development time, and the training time. From these metrics, a cost-benefit calculation can be done that estimates when it is feasible to use the new SRRS notation with its tool support. The experimental group is the test group that uses the SRRS notation. The control group is the test group that uses an updated version of the Thread-Capability notation.

1.4.6 Process to Formalize a Notation

Currently, a process is not available in the literature that describes how to take an existing notation and develop a formal one. As a result, selecting the steps involved in formalizing the existing notation is another challenge in this research. The challenge in defining the process involves determining which steps are essential in the process and then defining what needs to be done in the selected steps. The new process defined in this work has five steps in it. The steps are initially defined for the specific task at hand (i.e., defining the SRRS notation); however, the steps are also generalized so that the process may be re-used by other researchers.

1.5 Organization of Dissertation

This document is composed of seven chapters and four appendices. After this introduction, Chapter 2 provides background information on related work. The process that has been used
to develop the formal SRRS notation is described in Chapter 3. Chapter 4 presents the main features of the SRRS notation and is supported by the detailed specification of the SRRS notation in Appendix A. The features of the DSPEC notation are described in Chapter 5 with a detailed specification for the DSPEC notation in Appendix B. Chapter 6 summarizes the tool support development for the SRRS and the DSPEC notation. The experimental evaluation of the SRRS notation is summarized in Chapter 7. The detailed experimental design and results are in Appendix C. Finally, the conclusions and future work are in Chapter 8. Examples of specification units written in the SRRS notation are available in Appendix D.
Chapter 2  Related Work

2.1  Introduction

This chapter provides a summary of related requirements specification work. The chapter is organized around a taxonomy of requirements notations (refer to Figure 2.1). The requirements elicitation notations must be understandable to the customer. This category includes natural language techniques. The requirements specification and analysis notations lean toward precise, unambiguous, and concise notations, such as Z, higher order logic, or tabular specifications.

![Figure 2.1 A Taxonomy of Requirements Notations and Selected Examples]

2.2  Natural Language Techniques

Using natural language to describe requirements has two major advantages: there is no new notation to learn and natural language (e.g., English) offers a rich vocabulary and flexible
Chapter 2  Related Work

grammar. Customers who review requirements are generally comfortable with natural language.

For small requirements specifications a natural language specification may be a good selection. However, for large systems, this option does not scale well. Different authors may use a different style of writing and a different vocabulary when writing hundreds of pages of text. This makes the requirements specifications more difficult to review. In addition, in practice, tool support is limited to spellchecking and grammar checking tools. Defects in requirements written natural language are detected in a manual, peer review process.

An interesting analysis technique that may be applied to natural language specifications is the software fault tree analysis (SFTA) technique [Cha88]. This technique has been developed to verify the safety of the implementation (code) for complex, real-time software. The first step in the analysis is to identify one or more safety hazards. A safety hazard is a set of conditions (i.e., state) that has an unacceptable risk of leading to an accident. For example, a hazard in a traffic light control system is the state in which the lights in a traffic intersection are green in all directions at the same time. Once a hazard is identified, the code for the system is examined. Sections of the code that may relate to the hazard are identified and analyzed. If the code for the system does not prevent the hazard (i.e., the hazard can occur), then the system is corrected. The success of applying the technique depends on the ability of the analyst. It is a form of structured walk through that forces the analyst to view the program from a different perspective than during the development of the code. Programmers during development tend to concentrate on how the program should behave. When using the fault tree analysis, the analyst focuses on how the system should not behave. Although the SFTA technique has been developed for the analysis of code, it could be applied in the requirements specification phase. For example, the hazard is identified and
then requirements related to the hazard are identified and analyzed to determine if they preclude the hazard from occurring. In this phase, the requirement analyst tends to focus on what the system should do. The application of fault tree analysis at this point in the SDLC would consider what the system should not do. Applying the technique earlier in the SDLC can assist in detecting and correcting problems in the requirements for a system. The expected result is a better quality requirements specification document and a reduction in the time and cost for developing the system.

2.2.1 Use Cases

The UML’s use cases are currently a popular requirements specification based on using natural language [Boo99]. The use cases are lightly structured. Each one has a title, the normal sequence and then the alternate course(s) of action. There is no suggested or template phrasing that is defined by the approach nor is there a data model supported in the use case approach. Since there are no constraints on the style, the approach does not promote a blackbox style of writing the use cases.

The use cases use an external partitioning of the requirements. Each task the user needs to perform is described in a use case. More specifically, the user’s tasks are described in concrete use cases. A concrete use case is triggered by an external event such as a request from the user to display data.

In large system specifications, it is likely that a common sequence of actions is described in multiple use cases. To remove the duplication from the use cases, a sequence of actions that is used in multiple places may be documented, reviewed, and maintained in a single abstract use case and referenced, or triggered, by multiple concrete use cases. An abstract use case is always
Chapter 2  Related Work

triggered internally.

Specifying the data requirements in UML can be achieved with a separate domain object model. This model is a graphic representation of the data and the relationships among the data from the user’s perspective. An object diagram in UML has the following symbols (refer to Figure 2.2):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>object name: class</td>
<td>rectangle represents an object</td>
</tr>
<tr>
<td>attribute, value</td>
<td>line represents a link between two objects</td>
</tr>
</tbody>
</table>

Figure 2.2 Object Model Symbols

In the domain object model, the object is a real life “thing” that the customer can easily recognize. A link is an instance of an association between two objects. The links can represent different kinds of relationships including:

- dependencies (one object “uses” another object)
- inheritance (one object “is derived from” another object)
- associations (one object “knows about” another object)

The associations may have multiplicities (cardinality). The varieties of associations include:

- simple composition (whole/part relationship)
• aggregation (variation on composition relationship - more constrained)

Many of the advantages of the use case approach are due to its use of natural language as the notation. The use case approach is perceived as being simple to learn and use. Training material that covers the basics of the approach is readily available in sessions that take approximately one day. The weaknesses of use cases stem from its use of natural language as the approach’s notation. Automated tool support is not available to detect defects or to automatically generate test specifications. As a result, all defects must be detected manually and all test case specifications must be generated manually. These manual processes are time consuming and prone to error. The use case approach does not impose a black-box style for the requirements specification and leaves the style of writing the requirements up to the author to choose. This may result in a wide variety of styles being used when a large system is being specified by 20 or more authors. This variety can make reviewing the requirements specification document more difficult.

2.3 Formal Methods

2.3.1 Z Notation

Z is a well know formal requirements specification notation that has been used to describe system requirements [Pot92]. It is a mathematical notation based on typed, predicate logic. Since the mathematical notation is not based entirely on ASCII characters, tool support that provides mathematical symbols is necessary to document a specification (e.g., LATEX).

Being a typed notation, Z allows a user to declare data types or constants (i.e., data objects). Automated tool support is available to perform typechecking on the specifications. This automated tool support allows the authors to detect and remove defects early in the SDLC, improving the quality and reducing the overall cost of development.
Chapter 2  Related Work

The basic unit of specification in Z is called a schema. It is a paragraph of the form:

\[
\text{Name\_Of\_This\_Schema} = [\text{Declarations}, \text{Predicates}]
\]

A schema may be viewed as the encapsulation of a list of declarations and some associated constraints that are expressed by a list of predicates.

Research involving the Z notation has taken two interesting directions that are related to this work. The first is its use in the automated generation of test specifications from a specification written in Z [Hie97]. The benefits of automating this currently manual process include the potential to reduce the time to write system level test specifications, improve the quality of the test specifications, and reduce, as a result, the overall cost of developing a system. Secondly, research involving the Z notation has also turned to developing tool support to encourage its use in industry. Tool support to assist the development and typechecking of requirements specifications [Vas93] as well as tool support for training specification authors in how to use Z [Mor93] are available.

The disadvantages of using the Z notation include that it is challenging and time consuming to learn and the notation is difficult to read and understand. This is of particular concern for customers who are not prepared to validate a requirements specification written in Z. A work around for this problem is to provide schema in Z and a natural language summary of the schema. This work around however introduces a new problem in the requirements specification. If there are two descriptions of a requirement, then the two descriptions are difficult to keep consistent as requirements are updated. When the descriptions become inconsistent, the question arises as to which requirement description is the one that is developed for the customer.
2.3.2 S Notation

S is a higher order logic notation developed at The University of British Columbia [Joy94]. It is described as a general purpose, ASCII based (i.e., S can be written using standard text editors) notation. Tool support for the S notation includes a consistency checker, integration with a model checker, and integration with a theorem prover using a tool that can translate S into the Higher Order Logic (HOL) system's meta-language.

S has been developed as a lightweight alternative to using the HOL system. The following reasons have been noted in [Day00]:

1. The infrastructure of the theorem prover is unnecessary for automated analysis
2. The training time to learn to use the theorem prover is significant. The HOL system is perceived as intimidating to a novice specifier
3. The results of using the HOL theorem prover may not be useful because of the difficulty in determining the source of the defect causing a proof to fail.

S and its tool support is simpler to learn than the HOL system; however, S is a shorthand mathematical notation. Although it uses ASCII characters instead of the mathematical symbols (e.g., forall x instead of \forall x), S is a higher order logic. The author must learn the underlying mathematical concepts of higher order logic in order to use the S notation. This can be challenging and time consuming. To further complicate learning this notation, there is no training material available for S.

2.3.3 Software Cost Reduction

Software cost reduction (SCR) is a tabular notation that has been developed by the Naval
Chapter 2  Related Work

Research Laboratory [Hei98a, Hei98b]. SCR is based on the tabular specification used to document the operational flight program of the United States Navy’s A-7 aircraft. SCR has been used to specify the requirements for a number of systems including a telephone switching network, the shutdown for a nuclear power plant, and the operational flight program of Lockheed’s C-130J aircraft. Specifications based on tables have the advantage that they are modular (one table can be used to describe one part of the system’s functionality) and scalable.

Underlying SCR’s tabular notation is a state machine model. The semantics of SCR have been defined and a collection of software tools for specifying and analyzing software requirements is available. Automated tool support for SCR includes a consistency checker and validation tools including a simulator and a model checker. The consistency checker is a static analysis tool that detects type errors, circular definitions, variable name discrepancies, and missing cases. When an error is detected, the SCR consistency checker provides feedback to the user by displaying the table containing the error and highlighting the error. The simulator can be used to assist in the validation of the specification. A series of scenarios can be executed to determine if the specification behaves the way the user intends it to. The model checker is a more recent addition to the SCR toolset. An SCR specification can be translated into the Process Meta Language (Promela) which is the input language of the model checker Spin [Hol97]. The user may invoke Spin within the toolset to determine that the specification satisfies properties of interest.

For widespread acceptance and use in industry, there is one concern with the SCR notation. Due to its lower level of abstraction, the training time to learn and use SCR effectively may be higher than a project manager is prepared to invest in the requirements engineering staff. For example, when specifying a table in SCR it is necessary to know how many rows and
columns there are in the table, the types of all the variables, and how the variables interact. The variables’ types and interactions are not likely to be known until later phases in the SDLC.

2.3.4 Requirements State Machine Language

Requirements state machine language (RSML) is a notation that is based on a finite state machine model [Lev94]. The notation is derived from Harel’s Statecharts [Har87] and is well suited for modelling reactive, process control systems. The testbed for developing RSML is the traffic alert and collision avoidance system (TCAS II). This is a real aircraft avoidance system that operates in an aircraft, independently of the air traffic control system. TCAS II provides traffic advisories and recommended escape maneuvers to avoid conflicting aircraft.

RSML retains many features of the statechart notation:

- Superstates. This allows states to be grouped and supports the principle of abstraction.
- AND decomposition. This allows two or more state machines to run in parallel. This reduces the size of the state machine model.
- State-machine array. This supports the description of identical state-machines that run in parallel. A specific machine is selected using an index number.
- Connectives. Conditional connectives are used when an event occurs that can move the machine out of one state and into one of two or more possible states based on the guard condition.

In addition, RSML has made these changes to Statecharts:
• Directed communication. In addition to the broadcast communication defined in state-charts, RSML has an additional point to point communication feature. Directed messages can be sent and received over uni-directional channels between component state machines.

• Events. RSML has two types of events: internal and external. The internal events are the same as the events defined in statecharts. These events are used to order the evaluation of a function within a component of a system. The new, external events support communication between different components of a system.

• Interface definitions. Each separately modelled component in an RSML specification has an interface definition. The interfaces for input and output variables are defined. The definition for each input variable includes the source and destination of the message, the triggering RECEIVE event and guarding condition, the mapping of the message field names and values to variable names and value, and any internally generated events that result from receiving the message. The output variables defined are the message source and destination, internal triggering event and guarding condition, the mapping of the output variable names and values to message field names and values, and the internally generated external SEND event.

• Component state machine. Each component is composed of three parts, separated by double (thick), solid lines. The top part contains the input variables, the middle part contains the finite state machine, and the bottom part contains the output variables.
Chapter 2 Related Work

• Transition definitions. Each transition definition is composed of the identification, location, triggering event, guarding condition, and output action. A description, traceability, and comment section may also be added.

• State transitions. AND/OR tables are used to describe the state transitions. The rows have the AND logical condition and the columns have the OR logical condition. AND/OR tables are considered to be readable and straightforward to review. Macros, or labelled AND/OR tables, are also available to allow the logic in an AND/OR table to be re-used.

• Simplified graphics. For complex, fully or almost fully connected state machines, a transition bus is included in RSML to simplify the graphical representation of the model. Labels are removed from the state machine to simplify the picture and page numbers are used to indicate where to find access information in the model.

• Timing. Timing functions are written as logical expressions in guarding conditions. In the specification of TCAS, only three temporal functions are needed: the value of a variable in a previous point in time, the truth value of a logical condition at some point in the past, and an implicitly generated event based on time.

Part of the TCAS II specification written in RSML has been manually translated into computation tree logic (CTL), the notation accepted by a model checker called the symbolic model verifier (SMV) [And96]. CTL is a branching time temporal logic. In CTL, propositional logic is extended with temporal operators that express how propositions change their truth value over time. SMV has been used to check properties of the translated TCAS II specification as well as some safety properties that are specific to the aircraft collision avoidance domain. Properties of
the CTL specification include transition consistency (checking for non-deterministic transitions), function consistency (checking for function cases that are not mutually exclusive), step termination (checking for a step that does not terminate due to a cycle of events under the transition relation), and the presence of references in an AND/OR table to variables that have not been defined.

Learning to use the basic features of RSML is straightforward for because it is an extension of the familiar extended finite state machine. However, if the model checking capabilities are needed, learning to use CTL and the model checker SMV is challenging and time consuming. In addition, the manual translation of RSML into CTL is time consuming and prone to errors.

Although RSML is well suited for describing reactive, process control systems, it is not a good choice for describing data intensive systems. There is no integrated data model in the RSML notation. The author is restricted to describing relationships between data objects and types with comments.

2.3.5 Promela

Promela is a verification modeling language that has been developed to describe distributed systems. It is the input language for the model checker Spin [Hol97]. The intended use of Spin is to verify process behavior, described in Promela, with respect to properties the model should hold. A verification is typically performed in a series of steps, with the construction of increasingly detailed Promela models. Each model can be verified with Spin under different types of assumptions about the environment (e.g., message loss, message duplications, message corruption, etc.). Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification in the subsequent, more detailed models.
Promela programs consist of variables, processes, and message channels. Variables are used to store the state of the model. Variables may be declared either locally or globally. The scope of a variable is global if it is declared outside all process declarations and local if it is declared within a process declaration. Their values may only be inspected or changed by a process.

Processes are global objects that describe behaviour. The body of a process declaration consists of a list of zero or more declarations of local variables and/or statements. The declaration of a process called A below contains one local variable declaration and a single statement: an assignment of the value 3 to variable state:

```proctype
proctype A()
{
    byte state;
    state = 3
}
```

A process is instantiated when it is run and terminates when its last statement has executed and any processes it spawns have terminated. An example of running process A is:

```init()
{
    run A();
}
```
Chapter 2 Related Work

Message channels are used to model the transfer of data from one process to another. A message may be declared either locally or globally. The state of a message channel can only be changed or inspected by processes. A message channel is a first-in-first-out queue. An example of a channel declaration is below. Here, a channel called qname is declared such that it can hold up to 16 messages of type short:

\[
\text{chan qname} = [16] \text{ of short}
\]

All communication between processes takes place via either messages or shared variables. Both synchronous and asynchronous communication are modeled using a general message passing mechanism. Every statement in Promela can potentially model delay: it is either executable or not, in most cases depending on the state of the environment of the running process.

Critical sections of the statements in a system can be described in Promela by pre-fixing the statements with the keyword “atomic”:

\[
\text{byte state} = 1;
\]

\[
\text{proctype A()} \{ \text{atomic} \{ (\text{state==1}) \rightarrow \text{state} = \text{state}+1 \} \}
\]

\[
\text{proctype B()} \{ \text{atomic} \{ (\text{state==1}) \rightarrow \text{state} = \text{state}-1 \} \}
\]

\[
\text{init} \{ \text{run A(); run B()} \}
\]

In this case the final value of state is either zero or two, depending on which process executes. The other process is blocked forever.

Promela is a verification modeling language. There are, for instance, no elaborate abstract
data types, no more than a few basic types of variable, and no data model that describes the relationships among the data types and objects. A verification model is an abstraction of a protocol implementation. The abstraction maintains the essentials of the process interactions, so that it can be studied in isolation. Once a model has been constructed, it becomes the basis for the construction of a series of "verification suites" that are used to verify its properties. To build a verification suite, assertions are added by the author to the model. The assertions formalize the invariant relations about the values of variables or about allowable sequences of events in the model.

Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. During simulation and verification spin checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to verify the correctness of system invariants, locate non-progress execution cycles, and verify correctness properties specified in next-time free linear temporal logic properties. Learning to use Promela and Spin can be challenging and time consuming because the author must learn the underlying mathematical concepts of temporal logic in order to use the notation and tools.

2.4 Hybrid Approaches

Structured analysis and pseudo-natural language approaches have been placed into the hybrid approach category. The pseudo-natural language approaches described here include the Thread-Capability technique (developed at Hughes Aircraft of Canada, Ltd.), Heninger's templates, and selected languages that are based on using a constrained subset of English grammar including Attempto Controlled English (ACE) and Newspeak.
Chapter 2 Related Work

2.4.3 Structured Analysis

Structured analysis (SA) is an established requirements specification technique. The original structured analysis and design technique [Ros77a, Ros77b] provides a simple graphical notation (boxes and arrows) and strategies for applying the notation to requirements specifications (a hierarchical, top down decomposition of diagrams). Textual annotations of the diagrams is allowed using footnotes (a note about a box or arrow) or metanotes (a note about the diagram as a whole).

The original SA has been extended to include data dictionaries, decision tables, and process specifications for the primitive level processes by [Dem79]. At this point in its evolution, the SADT is restricted to describing data flow diagrams and is not well suited for specifying real time systems. A real time extension to Demarco’s structured analysis technique is introduced by [Hat88]. A real time extension that incorporates entity relationship diagrams (ERDs) and the concept of partitioning requirements from an external point of view are introduced by [War85]. The real time extensions integrate a control model (using control flow diagrams and finite state machines) with the data flow model in a specification.

2.4.3.1 Data Dictionary

A data dictionary is used in requirements specification techniques to reduce the vocabulary used and to specify the components and structure of the data and control entities in a requirements specification. The structure is specified using Backus-Naur Form (BNF) notation. Although there are many variants and extensions of BNF, it essentially consists of the following notation (refer to Table 2.1)
Chapter 2 Related Work

following notation: |

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>the production symbol used to declare a production rule</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>the conjunction symbol</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>double quotation marks delimit literal strings</td>
</tr>
<tr>
<td>[]</td>
<td>square brackets delimit optional terms</td>
</tr>
<tr>
<td>()</td>
<td>parentheses group terms</td>
</tr>
<tr>
<td>{}+</td>
<td>one or more iterations</td>
</tr>
<tr>
<td>{}*</td>
<td>zero or more iterations</td>
</tr>
<tr>
<td>* *</td>
<td>asterixes delimit a comment</td>
</tr>
<tr>
<td>\</td>
<td>backslashes delimit a primitive term</td>
</tr>
</tbody>
</table>

Table 2.1 BNF Notation

In a data dictionary, the relationships among data are restricted to those that can be described as a composition relationship. Data values, or objects, are described using comments.

2.4.3.2 ERDs

The basic object that the ER model represents an entity, which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence (e.g., a particular person or book) or it may be an object with a conceptual existence (e.g., a company or job). Each entity has particular properties, called attributes that describe it. For example, an employee entity may be described by the employee's name, age, salary, and job. A particular entity has a value for each of its attributes. These attribute values that describe each entity become major part of the data stored in the database.

A database usually contains groups of entities that are similar. For example, a company
employing hundreds of employees may want to store similar information concerning each of the
employees. These employee entities share the same attributes, but each entity has its own value(s)
for each attribute. Such similar entities define an entity type, which is a set of entities that have the
same attributes. For most databases, many entity types can be identified. Each entity type is
described by a name and a list of attributes.

The entity type description is called the entity type schema, or intension, and specifies a
common structure shared by all of the entities of that type. The schema specifies the entity type
name, the name and meaning of each of its attributes, and any constraints that hold on the individ­
ual entities. The set of particular instances at a particular instances at a particular moment in time
is called an extension of the data type. The schema (intension) does not change often because it
describes the structure of individual entities. The extension, however, may change often in a
database. For example, each time an entity is added or removed from the entity type the extension
changes.

The relationships between entities can be described as relationship types or as specific
relationship instances. A relationship type R among n entity types E₁, E₂, ..., Eₙ is a set of associ­
ations among entities from these types. Each relationship instance rᵢ in R is an association of
entities, where the association includes exactly one entity from each participating entity type.
Each such relationship instance rᵢ represents the fact that the entities participating in rᵢ are related
together in some way in the user’s world.

Relationship types usually have certain constraints that limit the possible combinations of
entities participating in relationship instances. The constraints are determined from the user’s
world. Two main types of relationship constraints, cardinality ratio and participation, are the structural constraints of a relationship type.

The cardinality ratio constraint specifies the number of relationship instances that an entity can participate in. Common cardinality ratios for binary relationship types are 1:N, 1:1, and M:N. The WORKS_FOR binary relationship type DEPARTMENT:EMPLOYEE is of cardinality ration 1:N, meaning that each department entity can be related to numerous employee entities (numerous employees work for a department) but an employee entity can be related to only one department (an employee works for only one department).

The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the entity relationship type. There are two types of participation constraints, total and partial. The total participation of an entity means that every entity in the set participates in the relationship. For example, in the WORKS_FOR relationship the participation of the EMPLOYEE entity is total because every employee works for a department. The partial participation of an entity means that a subset of the entities participate in the relationship. For example, in the MANAGES relationship, only some of the employees manage a department.

An alternate way of specifying the structural constraints is to associate a pair of integer numbers (min, max) with each participation of an entity type E in a relationship type R, where 0<=min<=max and max >=1. The numbers mean that for each entity e in E, e must participate in at least min and at most max relationship instances in R at all times. In this method min=0 implies partial participation where min>0 implies total participation. Using this method, structural constraints are simple to specify for relationship types of any degree.
An ERD is a graphic representation of the schemas (entity types and their relationships) for a database (refer to Table 2.2).

<table>
<thead>
<tr>
<th>ERD Graphic Symbol</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A rectangle</td>
<td>A rectangle represents an entity type</td>
</tr>
<tr>
<td>A double rectangle</td>
<td>A double rectangle represents a weak entity type</td>
</tr>
<tr>
<td>A diamond</td>
<td>A diamond represents a relationship type</td>
</tr>
<tr>
<td>A double diamond</td>
<td>A double diamond represents an identifying relationship type for weak entity</td>
</tr>
<tr>
<td>An oval</td>
<td>An oval represents an attribute (not a key)</td>
</tr>
<tr>
<td>An oval with a solid underlined name</td>
<td>An oval with a solid underlined name represents a key attribute</td>
</tr>
<tr>
<td>An oval with a dotted underlined name</td>
<td>An oval with a dotted underlined name represents a partial key</td>
</tr>
<tr>
<td>A double oval</td>
<td>A double oval represents a multivalued attribute</td>
</tr>
</tbody>
</table>

Table 2.2 ERD Symbols
<table>
<thead>
<tr>
<th>ERD Graphic Symbol</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Composite Attribute" /></td>
<td>A composite attribute is represented by two or more attributes</td>
</tr>
<tr>
<td><img src="image" alt="Derived Attribute" /></td>
<td>A dotted oval represents a derived attribute</td>
</tr>
<tr>
<td><img src="image" alt="Relationship" /></td>
<td>A line between an entity type and an attribute represents the relationship between the entity type and the attribute</td>
</tr>
<tr>
<td><img src="image" alt="Partial Participation" /></td>
<td>A single line between an entity type and a relationship type represents a relationship with partial participation</td>
</tr>
<tr>
<td><img src="image" alt="Total Participation" /></td>
<td>A double line between an entity type and a relationship type represents a relationship with total participation</td>
</tr>
<tr>
<td><img src="image" alt="Cardinality Ratio" /></td>
<td>A cardinality ratio (1:1, 1:N, M:N) is represented by a cardinality label</td>
</tr>
</tbody>
</table>

Table 2.2 ERD Symbols

Although SA may not be considered as a popular, up to date technique, research in the area continues. More recently, research in SA has been moving toward the use of formal methods. The formalization of the data flow diagrams has been worked on by [Pet98]. This work is focussed on defining the semantics of the transformation schema. Other work is focussed on
translating the SA model into a formal notation (e.g., Z) and then performing consistency checking between data and process diagrams [Kou96]. One limitation of this work is that the translation is not fully automated. The process specifications are translated manually into Z, a time consuming and error prone activity.

2.4.4 Pseudo-Natural Language Approaches

2.4.4.1 Templates

A technique described in [Hen80] introduces a straightforward, template driven approach that had been designed to specify hardware interfaces for embedded software. An example of a template is:

angle (?) is measured from line (?) to line (?) in the (?) direction, looking (?).

The templates look like natural language sentences. The parameters are interspersed in the template as they are in natural language. The user needs to fill in values to replace the question marks in the template. For example, the template provided above may be filled in like this:

magnetic heading is measured from the line from the aircraft to magnetic north to the horizontal component of the aircraft X axis, in the clockwise direction, looking down.

The informal templates improve the consistency and completeness of the interface specifications that are written by different authors. As the templates do not have to followed exactly, the application of the templates may vary. Manual reviews of the templates are performed, as no tool support is available. There is no training material available for the template approach.

2.4.4.2 Thread-Capability

The Thread-Capability technique has been developed by Hughes Aircraft of Canada, Ltd.
as their in house requirements specification notation for the CAATS and MAATS projects. A thread is a specification unit that specifies the actions performed by the system as the result of one or more stimuli. The thread specification unit is built on the concept of a path through the system which connects an external event or stimulus to an output event or response [Deu88]. The threads, or tasks, are straightforward to identify with domain expert's assistance because they describe what tasks the user needs to do. The threads are similar in their purpose as a concrete use case. A capability is like an abstract use case, in that it is a re-use mechanism and is triggered internally.

A significant difference between the use case approach and the Thread-Capability technique is that the threads are constrained by the template phrasing defined in the notation which promotes a "black box" point of view.

The Thread-Capability approach is a structured, natural language specification approach. A thread is composed of six sections: title; overview; stimulus; response; requirements; and performance. These are briefly described below.

1. The title provides a unique identifier for the thread in the specification document. The title is a short, meaningful description of the thread.
2. The overview section contains a high level description of the acceptance processing the thread provides, which is the processing performed if the stimuli are valid.
3. The stimulus section describes the external stimuli that trigger the thread. They are grouped into broad classes based on their functionality; all of the stimuli in a category are processed identically in the requirements section. Each group is associated with a local stimulus name which is referred to in the requirements section.
4. The responses section describes the output events that are generated by the thread. Like the
stimuli, the responses are grouped into categories based on their functionality. Each group is associated with a local response name, which is used in the requirements section.

5. The requirements section describes the functional requirements that relate to the thread. Each requirement specifies the processing done on a class of stimuli to generate the corresponding class of responses.

6. The performance section describes the mean and maximum response time categories for each stimulus-response pair in the thread. There are seven categories defined for the semi-formal threads.

A key component of the Thread-Capability specification approach is the data dictionary. It consists of a list of terms that describe the data required by all of the threads for the system under development. The data dictionary is a global, centralized repository. This approach is well suited for small to medium sized projects. For large projects, however, it is more difficult to maintain a data dictionary that is consistent (the definitions do not contradict), correct (the definitions are right), unambiguous (the definitions only have one interpretation), complete (all of the data definitions required are present), necessary (there are no extraneous definitions present), and up to date.

There are writing style conventions defined which provide a systematic and standard way of writing the threads. This improves the consistency of the threads written by different authors. The approach, however, is a natural language approach. The threads are difficult to parse unambiguously and analyze for inconsistencies, omissions, or extraneous information. The semi-formal thread approach relies on manual reviews of the threads to detect and correct errors. There is no tool support or training material available for Thread-Capability technique.
2.4.4.3 Attempto Controlled English

The specification language Attempto Controlled English (ACE) is a subset of English that has been defined for writing requirements specifications [Fuc99a]. ACE combines the familiarity of natural language with the rigor of formal languages. The underlying logic for ACE is first order logic. As a result, typechecking specifications written in ACE is not possible. However, parsing and translating the specification to Prolog has been accomplished.

The basic construct of an ACE specification is the declarative sentence. A declarative sentence is composed of a subject, a finite verb, and an optional complement or object. The declarative sentences may be combined with constructors to write composite sentences. The composite sentences are built from simpler sentences with the help of constructors that mark coordination (and, or, either-or), subordination (if-then, who/which/that), negation (not), and negated coordination (neither-nor). An example of an ACE specification is:

The customer enters a card and a numeric personal code. If it is not valid then the system rejects the card.

An ACE specification that conforms to the grammar is parsed deterministically. The tool support paraphrases the specification and reports the paraphrased specification back to the user. The user can decide if the parsing is what they intended to specify. If it is not what they intended, the user must re-write the sentence to obtain the desired parsing.

ACE specifications are machine processable. A specification may be parsed and/or translated into a knowledge base using Prolog. Since ACE is based on the concepts of English grammar, it is not a typed system. The following sentences are valid in ACE and would parse without an error:
Chapter 2 Related Work

The cow enters a card and a numeric personal code. If it is not valid then the system rejects the card.

This kind of type error needs to be detected in a manual, peer review.

If the specification is translated into Prolog, the knowledge base may be queried. This feature may be used to validate the requirements. A manual is available for the ACE notation [Fuc99b].

2.4.4.4 Newspeak

The Newspeak notation in [Osb96] uses an extensive subset of English grammar as a foundation. The Alvey Natural Language toolkit (ANLT) has been selected in the Newspeak work as a basis in order to start with a controlled language that covers as much of the language used to construct SRSs as possible. The ANLT contains one of the broadest grammars in existence and contains 40,000 entries in the lexicon. The selection of this grammar has been based on the idea that users of the Newspeak notation do not have to spend much time learning to use the controlled language. Instead, users can concentrate on the task of creating a readable SRS.

Tool support for the Newspeak notation includes a parser. When one or more possible parses are detected, the top five choices are presented to the user. The parses that are presented to the user are ranked by the tool as having the highest probability of being the correct parse. There is no training material available for Newspeak.

2.5 Summary of Approaches

There are numerous options to select from in choosing a notation to document a requirements specification. The techniques discusses in this chapter are summarized in Table 2.3. The
technique include well established approaches that are used in industry as well as approaches that are being researched. The characteristics summarized include their readability (i.e., does the notation look like natural language), the availability of tool support for detecting defects and automating the generation of test specifications, the support for a black-box style, the presence of an integrated data model, and the availability of training material.

2.6 Conclusions

Natural language (uncontrolled) has the advantages of being readable but lacks tool support to assist requirements author in detecting and correcting defects in the specifications. On the other hand, formal notations support a variety of automated tool support including parsers, typecheckers, model checkers, and theorem provers. However, formal methods using a mathematical shorthand are not considered readable and require a significant training investment. The hybrid notations are attempts to bridge these two extremes. Ideally, a controlled natural language notation could be developed that has a black-box style and has automated tool support for detecting defects and generating test specifications. The tool support can assist the author in developing a requirements specification of high quality in less time. In addition, the new notation needs to support an integrated data model in order to describe data intensive systems.
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>SCR</th>
<th>Z</th>
<th>S</th>
<th>Promela</th>
<th>Rsml</th>
<th>Thread-Capability</th>
<th>Newspeak</th>
<th>ACE</th>
<th>Templates</th>
<th>Use Cases</th>
<th>SA</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Looks Like Natural Language</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Automated Grammar Checking</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Automated Typechecking</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Automated Ump to Model Checker</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Automated Input to Theorem Prover</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Automated Test Case Specification</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Black-box Style</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Integrated Data Model</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Training Material</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 2.3 Requirements Specification Notations
Chapter 3  Formalization Process

3.1 Introduction

The process to create the formal SRRS notation is described in this chapter. The process begins with a semi-formal notation called Thread-Capability that has been used at Raytheon Systems Canada, Limited (Raytheon). The Thread-Capability notation has been used successfully on the CAATS and MAATS projects to document the software requirements specifications. These large scale projects have contracts valued at 500 million and 73 million Canadian dollars respectively [Aud96, Ray00]. The software requirements specification for the CAATS project is documented in approximately 3100 pages. The Thread-Capability notation is evaluated and an updated, semi-formal notation is developed called semi-formal SRRS. The updated notation addresses some of the problems identified in the original notation. At this point in the process the updated notation is still a semi-formal notation, as its syntax and semantics are not formally defined. The semi-formal SRRS is then evaluated. In this evaluation, the error checks that needs to be done by the notation are the focus. A sample requirements specification is written, reviewed, and corrected. The number and types of different errors are collected and four categories of defects are created. At this point, the notation is ready to be formalized by defining its syntax and the semantics. The error checks in the notation are also defined in this step.

3.2 The Formalization Process

The process used to formalize the Threads-Capabilities notation has five steps. Each of the steps is described in the sections below. The process is illustrated in Figure 3.1.
3.2.1 Evaluate Thread-Capability Notation

In order to improve the notation, it is necessary to evaluate its strengths and weaknesses [Coo97a]. The Thread-Capability notation's main strengths include:

- external partitioning of the requirements
Chapter 3 Formalization Process

- blackbox style
- abstraction, or grouping, of the stimuli and responses
- readability of the notation
- highly structured format
- definition of terminology with a data dictionary

The Thread-Capability notation has a number of drawbacks, however. These disadvantages include:

- lack of tool support to assist the authors in detecting defects
- difficulty in describing complex logical conditions
- lack of tool support to automate the development of system level (requirements based) testing
- lack of clearly defined matching rules used to pair a stimulus with a response
- lack of clearly defined re-use mechanism
- lack of training material

3.2.2 Draft Semi-formal SRRS Notation

The second step in the process is used to create the semi-formal SRRS notation. Two of the problems identified in the Thread-Capability notation are corrected in this step. The first correction is to clearly define the matching rules used to pair a stimulus with a response. Without these rules, the pairing of a stimulus with a response is ambiguous because different authors may each invent their own set of rules. Six matching rules are defined in the semi-formal notation. These rules are retained in the formal SRRS notation (refer to section 4.5). The second correction is to clearly define a re-use mechanism for the requirements. In large systems, a set of require-
ments may be used in multiple tasks. An example of a set of re-used requirements is authenticating a user's identify (e.g., confirming the user identification and password). In the semi-formal SRRS notation, a set of signalling messages are defined that allow the author to clearly identify the requirements being re-used. This mechanism is retained in the formal SRRS notation.

3.2.3 Evaluate Semi-formal SRRS Notation

In the third step of the formalization process, the SF SRRS notation is evaluated [Coo97b]. The set of error checks the formal version of the notation should support are identified in this step. The evaluation of the semi-formal notation begins with the authors, engineers at Raytheon, writing a non-proprietary specification in the semi-formal notation. In four iterations, the specification is reviewed and updated, and the data of interest is collected. The data collected includes the number and type of errors as well as the amount of time spent working on the task. Based on the data collected, an estimated 89% of errors could be automatically detected with tool support.

This evaluation led to describing four categories of errors including parsing, typechecking and type inference, analysis errors I, and analysis errors II. The categories increase in their level of difficulty of detecting with automatic tool support. Detecting the parsing errors is the simplest to automate, while detecting the analysis errors II defects is the most difficult.

Parsing errors are caused when a specification contains an illegal sequence of tokens, as defined by the grammar of the notation [Aho77, Eld94].

Typechecking and type inference errors are categorized together, as they are often intermingled in the static checking process [Lou93]. An example that describes both typechecking
and type inference is the application of a function or a predicate. In the application, the types of the actual parameters for arguments are checked to see if they match the types of the formal parameters (typecheck). If they do not match, then a typechecking violation occurs. A type inference error occurs when the actual result type of the application doesn't match the expected, or inferred, type (type inference). This may occur, for example, when a function application is used as an actual argument in another function application.

The analysis error I category contains error checks that indicate inconsistencies across sections of the specification that are straightforward to detect with automated tool support. Examples of these errors include types, constants, stimuli, or responses that are declared but not used and missing declarations.

The analysis error II category includes errors that are not straightforward to detect with automated tool support. Examples of these errors include duplicated functions or predicates with different names, missing requirements, and superfluous requirements. These errors are manually detected in peer reviews, or inspections, of the requirements specification document.

### 3.2.4 Formalize Notation

In this step, the syntax of the semi-formal notation is formalized, the semantics of the semi-formal notation are formalized, and the error checks identified in the previous step are defined in the notation. Refer to Chapter 4 for an overview of SRRS and to Appendix A for a detailed description.

The syntax of the SF SRRS is formalized by describing it in BNF. In order to determine if the syntax can be implemented, the tool support is developed at this point. A scanner and parser
Chapter 3  Formalization Process

are developed in lex and YACC.

To define the semantics, the SRRS notation is translated into S, a higher order logic. S is the logic used in the FormalWare project. The first task is to determine what the S notation provides. An S specification consists of a sequence of paragraphs. There are four kinds of paragraphs:

- **Type Declaration.** A type declaration paragraph is a statement that introduces one or more new types of data objects. For example, the type declaration below introduces the data types book and borrower.

  :book, borrower;

- **Constant Declaration.** A constant declaration paragraph is a statement that is used to introduce a new constant. In the S notation, a constant declaration can introduce a new data object, a function, or a predicate declaration. The constant declaration below introduces a new data object, Joe, that has the type borrower.

  Joe:borrower;

- **Type Definition.** A type definition paragraph is a statement that introduces a new type of data object and constants of the new type. In the S notation, the statement may be a constructive expression that introduces the data type book_status and three new constants on_shelf, mending, and charged as below.

  "the book has been charged out to" : borrower ->bool;
Chapter 3  Formalization Process

:book_status:= on_shelf | mending | charged ;

Alternatively, the type definition paragraph may be a statement that is a function or predicate application. For example, a predicate is applied below:

"the book has been charged out to" Joe

- Constant Definition. A constant definition paragraph is a statement that introduces a new constant and introduces the definitional axiom for the new constant. For example,

add_2_dollars_to_fine fine := fine + 2;

Once the paragraphs in the S notation are understood, the next step is to manually translate the SRRS specifications into S specifications. The S paragraphs must pass typechecking using the FUSS tool. The S notation has a number of rules that must be adhered to if the translation is to successfully typecheck with the S tool support, FUSS. These rules are provided in the list below.

1. White space is used as a delimiter in S. In the translation from SRRS to S, whitespace in the identifier for a data type, for example, is replaced by an underscore.
2. S does not support the flex-fix notation. The function and predicate declarations and applications must be converted from flex-fix into pre-fix notation.
3. S does not support the declaration of subtypes. The subtypes are translated into S by declaring a type name for the subtype and declaring a function that related the subtype to the base type.
4. Every parameter in a function application must be bound to a constant in S. This means that the data types and subtypes must be bound when translating a function application into S. For example, a reference to the data type <type 1> in an SRRS function application is bound in the translation to the data object called unnamed_type_1 with base type <type_1>.
5. The logical conditions are only recognized in S when written in upper case (i.e., “OR”, “AND”, and “NOT”).

6. Integer constants do not need to be declared as data objects in S. For example, 42 is declared with the syntax:

   42;

   S uses the identifier “num” to represent integers.

7. S cannot create paths that relate one data type to another. FUSS can only typecheck the function applications that are explicitly provided. This means that for each indirect data reference the path must be created (by DSPEC or the author) and then translated into S.

   After the translations are done manually, algorithms are developed that can be used to automate the procedure. The initial algorithms developed are in [Coo97b]. For example, the algorithm to translate a function declaration from an SRRS specification in the flex-fix format into a constant declaration in S in the pre-fix format is described below. The flex-fix format means that the formal arguments may be interspersed freely within the body of the function. The pre-fix format means that the formal arguments are placed at the end of the function body.

   while not at the return type in the SRRS function declaration
     if body of declaration present
       add body to S constant declaration
     else if parameter list present
       for each argument in list
Chapter 3  Formalization Process

add the formal argument to the body of the S constant declaration

add the formal argument to a list of formal arguments for the constant declaration

save the return type

for each formal argument in list

add the formal argument to the S constant declaration

add the return type to the S constant declaration

A symbolic example of an SRRS predicate declaration and its translation into a constant declaration in S is in Table 3.1

<table>
<thead>
<tr>
<th>SRRS Predicate Declaration</th>
<th>S Constant Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) “This is a flex-fix predicate declaration because the formal arguments (&lt;type 1&gt;) and (&lt;type 2&gt;) are intermingled with the body of the predicate” is a predicate.</td>
<td>“This is a flex-fix predicate declaration because the formal arguments &lt;type 1&gt; and &lt;type 2&gt; are intermingled with the body of the predicate” : &lt;type 1&gt; -&gt; &lt;type 2&gt; -&gt; &lt;bool&gt;;</td>
</tr>
</tbody>
</table>

Table 3.1  Translation of a Predicate Declaration in SRRS into a Constant Declaration in S

The categories of errors developed in the evaluation of the semi-formal notation are used as the basis for defining the error checks in SRRS. A total of 162 error checks are defined in the SRRS notation.

3.2.5 Evaluate Formal Notation (lab setting)

The formal, SRRS notation is ready for evaluation at this point in the process. There are several ways to evaluate a technique including case studies, pilot projects, or experiments. In this work, an experimental evaluation is selected to objectively evaluate the SRRS technique.

The purpose of this experiment is to objectively evaluate the costs and benefits of using a formal version of the SRRS notation along with its tool support in comparison to its semi-formal
version. The costs of introducing a formal notation include the cost of training employees in the tools and in the formal notation. To measure the costs, the amount of time spent in classroom and on the job training is recorded. The benefits include the availability of tools to assist the authors in automatically parsing, typechecking, and analyzing the specifications. The tool support is expected to reduce the time to develop the specifications and improve their quality. To measure the benefits, the amount of time used to write, review, and correct the specifications is recorded in addition to the number and type of defects recorded in the peer review process. In summary, the two techniques are compared in terms of the quality of the specifications written (number and category of defects detected) and the effort required to write the specifications (training, writing, reviewing, and correcting specification units). More rigorously, the objectives of the evaluation are to test the following hypotheses (Ho is the null hypothesis, Ha is the alternate hypothesis):

1. Ho: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has the same average detected defect rate per allocated requirement object than a notation that does not use a completely defined syntax and automated tool support.

   Ha: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has a lower average detected defect rate per allocated requirement object than a notation that does not use a completely defined syntax and automated tool support.

2. Ho: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has the same average effort per allocated requirement object to describe than a notation that does not use a
completely defined syntax and automated tool support.

Ha: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has a lower average effort per allocated requirement object to describe than a notation that does not use a completely defined syntax and automated tool support.

3. Ho: The use of a notation with a completely defined syntax and automated tool support results in the same average training time per subject than a notation that does not use a completely defined syntax and automated tool support.

Ha: The use of a completely defined syntax and automated tool results in a higher average training time per subject than a notation that does not use a completely defined syntax and automated tool support.

The average detected defect rate is defined in terms of the total number of detected defects in the specification units divided by the total number of requirement objects allocated to the specification units written.

The average effort per allocated requirement object identifier (ROID) is the total amount of time in minutes used to write, review, and correct the specification units written divided by the number of requirement objects allocated to the specification units written.

The average training effort per subject is the amount of time in minutes used to train the subjects in the notation both formally (in the classroom) and informally (independent study) divided by the number of subjects.
Chapter 3 Formalization Process

The results are very encouraging and indicate the SRRS technique is a cost-effective way to develop requirement specifications. The time to write, review, and correct the specification is reduced as are the number of defects detected in a peer review process. There is, however, an increase in the training time for the authors. A summary of the experimental design and results is provided in Chapter 7 and the details are available in Appendix C.

3.3 Conclusions

The formalization process used in this work is a systematic, step by step process that begins with a semi-formal notation, Thread-Capability, that has been used on large scale air traffic control projects at Raytheon Systems Canada Limited. The notation is updated using this process to correct two problems identified with the original notation. The updated notation is called semi-formal SRRS. An evaluation of semi-formal SRRS leads to creating four categories of defects including parsing, typechecking and type inference, analysis errors I, and analysis errors II. In the next step the semi-formal notation is formalized (syntax and semantics are defined) and its error checks are defined. The SRRS notation has 162 error checks. The advantage of defining the error checks is that defects in the specifications can be detected and reported to the user. The user can correct the problems and prevent them from being propagated into the next phases. The advantage of formalizing the notation is that the definition of the syntax and semantics supports the automatic translation of requirements into test case specifications. The requirements in SRRS can be translated in an S specification that can be used as the input to an automated test case generation tool. The test case generation tool has been developed as part of the work in [Don98]. This automatic translation is expected to reduce the amount of time and the cost of developing test cases by changing the step from a manual step to a step supported by a tool. The quality of the test case specifications is also expected to improve because errors are not
introduced using the automated process as they may be when the translation is done manually.

The five steps described in this process have been tailored and described for this problem, however, they can be generalized and re-used for formalizing a different semi-formal notation. Refer to Figure 3.2 for the generalized process.

Figure 3.2 A Five Step Process to Formalize a Notation
Chapter 4  Stimulus Response Requirements Specification Notation

4.1  Introduction

The SRRS notation is intended primarily for the specification of the software requirements for large, software intensive systems that have complex data requirements. For the purposes of this work a system is considered large when it has at least several thousand distinct requirements and is documented by a team of 20 or more authors over the course of a year or more. The data complexity may be characterized by a large number of related data definitions.

The SRRS notation is introduced in this chapter. The detailed specification for the syntax and error checking are in Appendix A. The evolution and characteristics of the SRRS notation are described in Section 4.2. The components of a specification unit are discussed in Section 4.3. A sample specification is also provided in this section to illustrate the syntax and rules of SRRS. Section 4.4 describes in more detail the three kinds of requirements that are used in the SRRS notation. The stimulus response rules are provided in Section 4.5. The relationship between SRRS and DSPEC is described in Section 4.6. Section 4.7 maps SRRS to S, a higher order logic. The translation of SRRS into S supports the automatic generation of test specification. An example of a specification unit written in SRRS, translated to S, and the test specifications automatically generated is provided in Section 4.8. Conclusions are in Section 4.9.

4.2  Evolution and Characteristics of SRRS

The SRRS notation has evolved from a long history of requirements specification notations (refer to Figure 4.1). The definition of the SRRS notation uses ideas and methods from over 20 years worth of requirements specification work. The important characteristics of the
Chapter 4 Stimulus Response Requirements Specification Notation

- external partitioning of requirements
- use of defined syntax, format
- requirements described as stimulus, processing, response
- generalized descriptions of data relationships
- machine readable, automated tool support

- external partitioning of requirements
- use of data dictionary (BNF)
- use of highly structured English
- requirements described as stimulus, processing, response

- external partitioning of requirements
- requirements described in terms of inputs, processing, outputs

Templates [Hen80]


ERDs [Chen76]

Z [Spi88]
HOL88 [Gor93]

Structured Analysis [Dem79]
- use of data dictionary (BNF)
- use of structured natural language

Structured Analysis [War85]
- external partitioning of requirements
- data schema ERDs

Structured Analysis [Ros77a, Ros77b]
Software Cost Reduction (SCR) Tables
Survey paper [Dav77]
- requirements described in terms of inputs, processing, outputs

Templates [Hen80]


Use Case [Jac92, Boo99]
- external partitioning of requirements
- natural language
- formatting constraints

Figure 4.1 Evolution and Influences on the SRRS Notation
notation and where they have evolved from are described in this section. The characteristics include: partitioning of the requirements; stimulus response (blackbox) style; abstraction of inputs and outputs; structured natural language; and a means of expressing complex data relationships and logical conditions.

### 4.2.1 Partitioning of Requirements

Two broad categories for partitioning the software requirements for a system are “top-down” and “outside-in”[War85]. The “top-down” decomposition is implementation dependent because an arbitrary choice of many possible decompositions is made. The “outside-in” approach uses the stimuli and responses to and from the system to partition the software requirements. The sources and sinks may be either human or other systems. An advantage of employing the “outside-in” partitioning is that the resulting specification is well suited to use as a communication tool with the end user of the system. The specification is organized around the input and outputs of the system, which corresponds to a user’s point of view. The “outside-in” partitioning is used in structured analysis [War85], use cases [Jac92, Boo99] and Thread-Capability [Hug93].

The SRRS technique applies the “outside-in” partitioning to structure the software requirement specification document. As a result, the documentation is straightforward to review, especially with respect to the scope of the system’s capabilities.

### 4.2.2 Stimulus Response (blackbox) Style

A blackbox description of software requirements describes the behaviour of the system in terms of its external stimuli (inputs) and external responses (outputs). In general, every require-
Chapter 4 Stimulus Response Requirements Specification Notation

ment is specified in terms of a relationship between an external stimulus and an external response. The advantages of using a blackbox approach for describing requirements include minimizing the potential for including internal design details in the specification and maximizing the suitability of the specification for testing the system's software. Discouraging the inclusion of design details in the specification decreases the likelihood of overly constraining the design and makes the specification simpler to maintain as the design details may change as the project develops. Minimizing the software's internal processing descriptions simplifies the development of blackbox test cases as the test engineers do not have to derive the requirements based test cases from descriptions of internal processing. A stimulus response style of requirements specification is seen as an important characteristic for testability early in requirements specification research [Dav77]. This style is used in Thread-Capability [Hug93].

The SRRS notation uses the stimulus response style. The requirements are described in terms of a direct relationship between a group of externally generated stimuli and a group of externally visible responses. Because of the blackbox style, a specification written in the SRRS notation is well suited for use as a working document by the system test group. The blackbox approach also minimizes the inclusion of internal processing (design details) in the specification which simplifies the maintenance of the specification.

4.2.3 Abstraction of Inputs and Outputs

For systems with a large number of stimuli and responses it is convenient to group inputs (stimuli) and outputs (responses) and to be able to refer to these groups by name. This is an abstraction of the stimuli and responses. The use of group names supports the development of a specification which is concise and simple to maintain. For example, suppose there are six differ-
ent stimuli that trigger the same processing requirements in a system. If a name is given to this
group of six stimuli, the name can be used to describe triggering one requirement. The alternative
is to repeat the requirement for each of the six different stimuli that triggers the processing.

When groups of stimuli and groups of responses are related by a single requirement, rules
are needed to explicate which stimulus is matched with which response. For example, if a group
name represents six stimuli and a second group name represents six responses, then the notation
must have rules to determine which of the 36 possible requirements are represented. Grouping
inputs and outputs is used in Thread-Capability [Hug93]. The Thread-Capability does not define
the stimulus response matching rules used to explicate the requirements.

The SRRS notation uses an abstraction of the input and output to support the development
of a concise and maintainable specification. The rules for matching the inputs and the outputs are
defined in SRRS.

4.2.4 Structured Natural Language

Structured natural language is used in structured analysis techniques [Dem79, War85] as a
mechanism to improve the consistency of specifications by introducing writing constraints. The
writing constraints reduce the vocabulary and restrict the grammar and style of the natural
language text written. The idea of restricting the grammar and style is extended in [Hen80],
[Hug93], and [Coo97a] as standard, template phrasing is provided for the authors to use. This
improves the consistency of the requirement specification document by providing a standard way
of describing a problem that is encountered repeatedly.

A further extension to structured natural language is the definition of a concrete syntax.
When this is done, tool support can be developed to assist authors in determining if the require-
ments conform to the syntax. The SRSS notation has a concrete syntax to improve the consistency of the specification written by a large group of authors. For example, the concrete syntax for one requirement statement is:

Upon receipt of a [group stimulus name], the system shall send a [group response name].

The bolded words or characters represent the standard part of the phrase. The plain type words are the parts of the sentence that are filled in by the author.

The syntax also restricts the verbs that are used in the responses to a set of five action verbs. For example, the verb send is used in the requirement above. The purpose of the reduced vocabulary is to ensure consistency among the requirements written by different authors. A second benefit of this writing constraint is that it discourages the inclusion of computer human interface (CHI) details. The grammar does not include CHI related terminology such as “display”, “double click”, or “enter”. The result is that the system is described in terms of a virtual environment, which isolates the requirements from the CHI design and the interface technology used to implement the CHI design. As CHI requirements may change frequently, the exclusion of CHI details from the specification makes the document simpler to maintain.

4.2.5 Describing Complex Data and Logical Conditions

4.2.5.2 Data Models

Data dictionaries and data schema are used to describe data, organize data, and describe how the data are related. The data repositories (data dictionaries or ERD models) encourage a reduction in the vocabulary. This restricts authors in their choice of words to describe the behaviour of the system. A well maintained data dictionary, for example, is one means to reduce the variety of terms used in a project. Ideally, data items are re-used by different authors, rather
than each author creating duplicate entries in the data repository. This improves the consistency of
the specification because all authors use the same terminology to describe the data in the require­
ments. Data dictionaries defined in BNF are used in [Dem79, Hug93] to reduce and standardize
the vocabulary in the specification. ERDs are used in [War85] to describe the data and their
relationships.

A variation on the standard BNF data dictionary or ERD data models is used in SRRS. The
variations include a change in the scope of the data repository and the notation used. The
SRRS notation uses a local data repository instead of a global, or project wide, repository. BNF is
not used because it is restricted to the “is composed of” relationship. A more general data model is
used in SRRS that has the flexibility of an ERD and is ASCII based.

4.2.5.3 Complex Logical Conditions

Tabular formats for describing complex, logical conditions are concise and readable. Require­
ments specification techniques use tables either as a stand-alone notation or integrate
them with other notations [Day98]. The format of the tables supported in the SRRS technique are
a variation of an AND/OR table [Day98]. The tables have the following format:

<table>
<thead>
<tr>
<th>Label</th>
<th>Case 1</th>
<th>Case 2</th>
<th>...</th>
<th>Case n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label 1</td>
<td>predicate</td>
<td>predicate</td>
<td>...</td>
<td>predicate</td>
</tr>
<tr>
<td>Label 2</td>
<td>predicate</td>
<td>predicate</td>
<td>...</td>
<td>predicate</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>predicate</td>
</tr>
<tr>
<td>Label m</td>
<td>predicate</td>
<td>predicate</td>
<td>predicate</td>
<td>predicate</td>
</tr>
<tr>
<td>Title</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The label of each row in the predicate table is an expression. The cells of each row are
predicates that are applied to the row’s label. Each column represents a single case, where the
conjunction of the cells in the column is true. Any other cases not described in the table explicitly are assumed to be false. The table represents an “if, else if” structure. The predicate table’s name is in the last row of the table.

The inclusion of a tabular format for describing complex conditions adds flexibility to the SRRS notation. The author has the choice of describing simpler conditions in the style that looks more like natural language and describing complex conditions in the tabular style. If a different tabular notation is preferred, then the tables in SRRS may be replaced in the future.

### 4.2.6 Tool Support and Training Material for SRRS

Tool support and training material have been developed for the SRRS notation. The tool support allows the authors of requirements specification units to check the format of their requirements, execute the error checking that is defined in the notation, or translate the specification into S. The S translation may be used as input to a test case generation tool to automatically generate test specifications. The tool support for SRRS is described in detail in Chapter 5.

Training material including lecture slides, user manual, and an introductory tutorial are available for SRRS [Coo99]. The lecture slides are designed for use in a one day training course. The course is composed of four modules and a hands-on exercise. The modules are organized as follows:

- **Module 1: Overview of the Software Development Lifecycle**
  - Software Development Lifecycles (SDLCs)
  - Introduction to Requirements

- **Module 2: Initial Phases of Software Development Lifecycle**
4.3 Overview of a Specification Unit Written in SRRS

A specification unit written in SRRS introduces a unique section of a software requirements specification document. A specification unit is highly structured and is composed of seven parts: title, overview, stimuli, responses, requirements, declarations, and a performance section (refer to Figure 4.2). The purpose of each of these sections is summarized below. The detailed specification of the notation is presented in Appendix A.
4.3.1 **Title Section**

The title provides a unique identifier for the specification unit in the requirements specification document. The title should be a meaningful name that indicates the contents of the specification unit. The title is not a testable requirement.

4.3.2 **Overview Section**

The overview section provides a summary description of the rejection and normal processing performed by the specification unit. There are no testable requirements in the Overview section.
4.3.3 Stimuli Section

The stimuli section documents all of the stimuli that trigger the specification unit. The stimuli section is organized as a list of unique group stimulus names. Each group stimulus name is the name for a list of (source, message) pairs or (source, message, message) triples. The source name indicates the origin of the stimulus message. The message indicates the data type of the stimulus message. The group stimulus name provides a name for referencing one or more of the elements of the list in the requirements section in a compact notation.

4.3.4 Responses Section

The responses section documents all of the responses that are generated by the specification unit. The responses section introduces unique group response name declarations. The group response name is the name for a list of one or more unique (destination, message) pairs or (destination, message, message) triples. The destination indicates where the response is going. The message indicates the contents of the response.

4.3.5 Requirements Section

The requirements section describes the processing required to transform a stimulus into one or more responses. The requirements section is composed of one or more requirement statements that are written in a particular order. The order of the requirements is rejection requirements, acceptance requirements, followed by general requirements. The rejection requirement describes the generation of a rejection with reason, for example, when a validation check fails in a requirement. The acceptance requirement describes the generation of an acceptance response. The purpose of the acceptance requirement is to document the requirement to provide feedback to the
user. A general requirement expresses either a relationship between a stimulus and one or more responses or the generation of one or more responses.

A general requirement can be extended by describing response qualification and/or response response requirements (refer to Section 4.4). A response qualification is either a predicate or a boolean condition. A response response requirement is a compact notation for additional stimulus response requirements (i.e., there is one response followed by another response). The interpretation of the stimulus response, response qualification, and response response requirements is presented in Section 4.4.

### 4.3.6 Declaration Section

The declaration subsections introduce data types, data subtypes, data objects (constants, variable system parameters, system state components), functions, and predicates. The functions and predicates are categorized as local (declared in this specification unit), external (declared in another specification unit), or exported (declared in this specification unit for use in other specification units). Error checking in the specification unit is with respect to the context described.

The context of a specification unit is composed of a type space and an object space. The type space contains the names of the data types and subtypes that the user declares in addition to the built-in data types. The object space contains the names and types of the data objects the user declares in addition to the built-in data objects. The data objects include constants, variable system parameters (VSPs), and system state components. Constants are domain independent data objects that have values that do not change, for example "TRUE", "FALSE", or the value of π. A VSP is a data object with a value that may be changed for the system in order to tailor the system for a specific installation. When a VSP is used in a specification unit, however, it is
considered to have a fixed value. An example of a VSP is "Maximum number of library items allowed for charge". The system state components are the domain specific data objects the author needs to use in the specification unit. For example, system state components are "on-shelf" or "in-mending".

4.3.7 Performance Section

The performance section describes the performance requirements for the specification unit as a logical condition.

4.4 Stimulus Response, Response Response, Response Qualification Requirements

The simplest requirement in the SRRS notation is a stimulus response requirement. The general structure documents the receipt of a stimulus and the generation of a response for a requirement. An example of a stimulus response requirement is:

1) Upon receipt of a [w request], the CSCI shall send an [x response]

An optional logical condition can be present on a stimulus response requirement. For example, a stimulus response requirement with the logical condition "an important condition holds" is:

1) Upon receipt of a [w request], if {an important condition holds}, the CSCI shall send an [x response].

A stimulus response requirement can be followed by one or more response qualification requirements. A response qualification is a postcondition that is associated with the response. The response qualification may be a data object of type bool, a subtype of bool, type bool, or a predicate application. For example, a stimulus response requirement with the response qualifica-
tion "The date in the response is set to the current date" is:

1) Upon receipt of a [w request], if {an important condition holds}, the CSCI shall send an [x response].
{The date in the response is set to the current date}.

A stimulus response requirement may be followed by one or more response response requirements. A stimulus response requirement followed by a response response requirement is a compact notation for two or more stimulus response requirements. There are four rules to apply in expanding the compact notation. The first rule is that the stimulus on the stimulus response requirement, if present, is added on to each of the response response requirements. The second rule is that the logical condition on each of the response response requirements, if present, is added to each of its subsequent response response requirements. The third rule is that the logical condition on the stimulus response requirement, if present, is added on to each of the response response requirements. The fourth rule is that response qualifications, if present, remain associated with their stimulus response or response response requirement. An example of applying these rules is presented below.

Before Applying the Response Response and Response Qualification Rules

1) Upon receipt of a [w request], if {an important condition holds}, the CSCI shall send an [x response].
{The date in the response is set to the current date}.
If the {{[x response]} is sent and {some condition holds}, the CSCI shall commit a [y response].If the {{[y response ]} is committed and {some other condition holds} , the CSCI shall update a [z response]. {The colour is set to green in the updated response}.

After Applying the Response Response and Response Qualification Rules

1)Upon receipt of a [w request], if {an important condition holds} , the CSCI shall send an [x response].{The date in the response is set to the
Chapter 4 Stimulus Response Requirements Specification Notation

current date).

2) Upon receipt of a [w request], if {{an important condition holds} and {some condition holds}}, the CSCI shall commit a [y response].

3) Upon receipt of a [w request], if {{an important condition holds} and {{some condition holds} and {some other condition holds}}}, the CSCI shall update a [z response]. {The colour is set to green in the updated response}.

Once the stimulus response requirements are generated the stimulus response matching rules are applicable (refer to Section 4.5).

4.5 Stimulus Response Matching Rules

The requirement statements relate a group of stimuli that trigger the processing to one or more groups of responses using a compact notation. The stimuli and responses must be matched, however, to explicitly represent all of the requirements. The rules for generating the stimulus response pairs are described here.

There are six cases to consider when matching the stimulus and response pairs. Cases one through four are for requirements that have a stimulus present. The logical condition is optional when the stimulus is present and does not affect the stimulus response matching rules. The cases shown below do not include a logical condition. Cases five and six are for requirements that do not have a stimulus present. The first case has no qualification on the source or destination. The second case has an unqualified source and a qualified destination. The third case has a qualified source and an unqualified destination. The fourth case has a qualified source and destination. The fifth case has an unqualified destination. The sixth case has a qualified destination.

Case One

Upon receipt of a [group stimulus name], the system
shall send a [group response name]

For each stimulus in the [group stimulus name]
Obtain the source name
For each response in the [group response name]
If the source name is not an internal signal
    If the destination name matches the source name,
        pair the stimulus and response
    pair the stimulus with the internal signals
    of the response
Else
    pair the stimulus and response
    pair the stimulus with the internal signals of
    the response

In case one, for externally visible stimuli, a stimulus is paired with every response that has the same destination name as the source name. For internally signalled stimuli, a stimulus is paired with every response.

Case Two
Upon receipt of a [group stimulus name] from X, the system shall send a [group response name].

For each stimulus in the [group stimulus name]
Obtain the source name
If the source name is X then
    For each response in the [group response name]
    If the source name is not an internal signal
        If the destination name matches the source name,
            pair the stimulus and response
        pair the stimulus with the internal signals
        of the response
    Else
        pair the stimulus and response
        pair the stimulus with the internal signals
        of the response

In case two, for externally visible stimuli, a stimulus with source name X is paired with every response with destination X and the associated internal responses. For internally signalled stimuli, a stimulus with source name X is paired with every response.
Case Three

Upon receipt of a [group stimulus name], the system shall send a [group response name] to Y.

For each stimulus in the [group stimulus name]
Obtain the source name
If the source name is Y then
   For each response in the [group response name]
      If the destination name is not an internal signal
         If the destination name matches the source name,
            pair the stimulus and response
            pair the stimulus with the internal signals of the response
      Else
         If the destination name matches the source name
            pair the stimulus and response

In case three, for externally visible responses, a response with destination name Y and the associated internal responses are paired with every stimulus with source Y. For internally signalled responses (i.e. the destination Y is an internal send/commit/update signal), a response with destination name Y is paired with every stimulus that has source name Y.

Case Four

Upon receipt of a [group stimulus name] from X, the system shall send a [group response name] to Y.
For each stimulus in the [group stimulus name]
Obtain the source name
If the source name is X then
   For each response in the [group response name]
      If the destination name Y is not an internal signal
         If the destination name is Y
            pair the stimulus and response
            pair the stimulus with the internal signals of the response
      Else
         Else
If the destination name is Y
pair the stimulus and response

In case four, if Y is an externally visible response, a response with destination name Y and the associated internal responses are paired with every stimulus with source X. For internally signalled responses (i.e. the destination Y is an internal send/commit/update signal), a response with destination name Y is paired with every stimulus with source X.

Case Five

If {logical condition}, the system shall send a [group response name].

For each response in the [group response name]
generate a stimulus response requirement
(stimulus is null)

In case five, a stimulus response requirement is generated for each response in the group response name.

Case Six

If {logical condition}, the system shall send a [group response name] to Y.

For each response in the [group response name]
If the destination name Y is not an internal signal
   If the destination name is Y
      generate a stimulus response requirement (stimulus is null)
generate a stimulus response requirement for the internal signals of the response
(stimulus is null)
Else
   If the destination name is Y
      generate a stimulus response requirement
      (stimulus is null)
Chapter 4 Stimulus Response Requirements Specification Notation

In case six, for externally visible responses, a stimulus response requirement is generated for each response with destination name Y and the associated internal responses. For internally signalled responses (i.e. the destination Y is an internal send/commit/update/signal), a stimulus response requirement is generated for every response with destination name Y.

4.6 SRRS and DSPEC

The SRRS notation uses DSPEC as a foundation, or core, language. SRRS is a separate language that is designed for use in the software requirements specification phase of the software development lifecycle. This notation allows the users (requirements authors) to express requirements about the relationships among data expressions and the logical conditions from a requirements specification perspective.

SRRS uses DSPEC by making requests to evaluate a logical condition or data expression. The logical condition or data expression is evaluated using the DSPEC context. The data objects, data types, functions, and predicates declared in the SRRS specification unit are translated into the DSPEC context. The translation from the SRRS context to the DSPEC language context is described in Table 4.1. Because DSPEC is designed to manage the logic and data complexity,

<table>
<thead>
<tr>
<th>SRRS Context</th>
<th>DSPEC Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>N/A</td>
</tr>
<tr>
<td>Overview</td>
<td>N/A</td>
</tr>
<tr>
<td>Stimuli</td>
<td>N/A</td>
</tr>
<tr>
<td>Responses</td>
<td>N/A</td>
</tr>
<tr>
<td>Requirement</td>
<td>N/A</td>
</tr>
<tr>
<td>Performance</td>
<td>Logical Condition</td>
</tr>
<tr>
<td>Logical Condition</td>
<td>Logical Condition</td>
</tr>
<tr>
<td>Data Expression</td>
<td>Data Expression</td>
</tr>
</tbody>
</table>

Table 4.1 Translating the SRRS Context to the DSPEC context
Chapter 4 Stimulus Response Requirements Specification Notation

<table>
<thead>
<tr>
<th>SRRS Context</th>
<th>DSPEC Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Name Declaration</td>
<td>Type Declaration</td>
</tr>
<tr>
<td>Subtype Declaration</td>
<td>Subtype Declaration</td>
</tr>
<tr>
<td>Static Association Declaration</td>
<td>Function Declaration</td>
</tr>
<tr>
<td>Constant Declaration</td>
<td>Name Declaration</td>
</tr>
<tr>
<td>Variable System Parameter Declaration</td>
<td>Name Declaration</td>
</tr>
<tr>
<td>System State Component Declaration</td>
<td>Name Declaration</td>
</tr>
<tr>
<td>Stimulus Response Component Declaration</td>
<td>Name Declaration</td>
</tr>
<tr>
<td>Local Function Declaration</td>
<td>Function Declaration</td>
</tr>
<tr>
<td>Local Predicate Declaration</td>
<td>Predicate Declaration</td>
</tr>
<tr>
<td>External Function Declaration</td>
<td>Function Declaration</td>
</tr>
<tr>
<td>External Predicate Declaration</td>
<td>Predicate Declaration</td>
</tr>
<tr>
<td>Exported Function Declaration</td>
<td>Function Declaration</td>
</tr>
<tr>
<td>Exported Predicate Declaration</td>
<td>Predicate Declaration</td>
</tr>
</tbody>
</table>

Table 4.1 Translating the SRRS Context to the DSPEC context

only the components from the SRRS notation that are related to this functionality are translated and subsequently used by the DSPEC language.

4.7 Translation to S

The SRRS notation can be translated to the typed, predicate logic S (refer to Table 4.2). The S output can be used to automatically generate test case specifications [Don98].

<table>
<thead>
<tr>
<th>Stimulus Response Requirement Component</th>
<th>S Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>N/A</td>
</tr>
<tr>
<td>Overview</td>
<td>N/A</td>
</tr>
<tr>
<td>Stimuli</td>
<td>N/A</td>
</tr>
<tr>
<td>Responses</td>
<td>N/A</td>
</tr>
<tr>
<td>Performance</td>
<td>Predicate Application or Boolean expression</td>
</tr>
</tbody>
</table>

Table 4.2 Translation of SRRS to S
### Chapter 4 Stimulus Response Requirements Specification Notation

#### 4.8 Library Specification Unit Example

The following specification unit is taken from a larger example for an Integrated On-line
Library System (IOLS).

4.8.1 Specification Unit

Title:
\Search Catalogue\ 

Overview:
The Search Catalogue specification unit describes the processing performed when an operator or external library requests to search for an item. The results of the search are sorted in alphabetical order.

Stimuli:
1) The IOLS shall satisfy the requirements described below upon receipt of a [search request] from:

   a) Operator <search request>;
   b) “Remote Operator” <search request>;
   c) VPL <search request>;
   d) SFU <search request>;
   e) UVIC <search request>;
   f) EPL <search request>.

Responses:
1) As specified by the requirements described below, the library system shall return an [acceptance] to:

   a) Operator <acceptance>.
2) As specified by the requirements described below, the library system shall send a [search response] to:

   a) Operator <search response>;
   b) "Remote Operator" <search response>;
   c) VPL <search response>;
   d) SFU <search response>;
   e) UVIC <search response>;
   f) EPL <search response>.

**Requirements:**

1) Upon receipt of a [search request], if { the input is not rejected}, the IOLS shall return an [acceptance].

2) Upon receipt of a [search request], the IOLS shall send a [search response]. {The search response is sorted alphabetically}.

**Declarations:**

**System State Components:**

1) "The search response is sorted alphabetically":<bool>.

2) "the input is not rejected":<bool>.

**Stimulus Response Components:**

1) <search request>:<request message>.

2) <search response>:<response message>.

3) VPL:<source>;
4) VPL:<destination>;
5) SFU:<source>;
6) SFU:<destination>;
7) UVIC:<source>;
8) UVIC:<destination>;
9) EPL:<source>.
10) EPL:<destination>.

Local Predicates:

1) “The response times for the I/O transactions described in this specification unit shall be in Class” (<integer>) is a predicate.

Performance:

1) {The response times for the I/O transactions described in this specification unit shall be in Class {2}}.

4.8.2 Translation to S

The S file is automatically generated with a request to the SRRS tool support. The S output is listed below.

```
% fuss session initialization
%init
%include startup.s

% type declarations
```
:integer;
:request_message;
:response_message;
:source;
:destination;
:hour;
:millisecond;
:second;
:minute;
:metre;
:millimetre;
:centimetre;
:decimetre;
:kilometre;
:day;
:month;
:year;
:date;
:day_of_year;
:day_of_week;
:day_of_month;
:celsius;
:litre;
Chapter 4  Stimulus Response Requirements Specification Notation

:millilitre;
:decilitre;

% subtype declarations
:Operator_source;
:Operator_destination;
:Remote_Operator_source;
:Remote_Operator_destination;
:internal_send_signal_source;
:internal_send_signal_destination;
:internal_commit_signal_source;
:internal_commit_signal_destination;
:internal_update_signal_source;
:internal_update_signal_destination;
:internal_signal_source;
:internal_signal_destination;
:external_commit_destination;
:external_update_destination;
:Printer_source;
:Printer_destination;
:Plotter_source;
:Plotter_destination;
:search_request_request_message;
% object declarations

TRUE_bool:bool;

FALSE_bool:bool;

The_search_response_is_sorted_alphabetically_bool:bool;

The_input_is_not_rejected_bool:bool;

unamed_Operator_search_request:source;

unamed_search_request_request_message:request_message;

unamed_Operator_acceptance:destination;

unamed_acceptance_response_message:response_message;

unamed_Operator_search_response:destination;

unamed_search_response_response_message:response_message;

unamed_Remote_Operator_search_request:source;
Chapter 4 Stimulus Response Requirements Specification Notation

```
unamed_Remote_Operator_search_response:destination;
unamed_VPL_search_request:source;
unamed_VPL_search_response:destination;
unamed_SFU_search_request:source;
unamed_SFU_search_response:destination;
unamed_UVIC_search_request:source;
unamed_UVIC_search_response:destination;
unamed_EPL_search_request:source;
unamed_EPL_search_response:destination;

% function declarations
The_response_times_shall_be_in_Class_integer_bool:integ
er -> bool;
Operator_source:source -> source;
Operator_destination:destination -> destination;
Remote_Operator_source:source -> source;
Remote_Operator_destination:destination -> destination;
internal_send_signal_source:source -> source;
internal_send_signal_destination:destination ->
destination;
internal_commit_signal_source:source -> source;
internal_commit_signal_destination:destination ->
destination;
```

82
Chapter 4 Stimulus Response Requirements Specification Notation

internal_update_signal_source:source -> source;
internal_update_signal_destination:destination -> destination;
internal_signal_source:source -> source;
internal_signal_destination:destination -> destination;
external_commit_destination:destination -> destination;
external_update_destination:destination -> destination;
Printer_source:source -> source;
Printer_destination:destination -> destination;
Plotter_source:source -> source;
Plotter_destination:destination -> destination;
search_request_request_message:request_message -> request_message;
search_response_response_message:response_message -> response_message;
acceptance_response_message:response_message -> response_message;
VPL_source:source -> source;
VPL_destination:destination -> destination;
SFU_source:source -> source;
SFU_destination:destination -> destination;
UVIC_source:source -> source;
%Requirements
Rl := if ( Stimulus_request_message_bool ( unnamed_Operator_search_request, 
unnamed_search_request_request_message ) AND 
The_input_is_not_rejected_bool) then ( 
return_response_message_bool ( ( 
unnamed_Operator_acceptance , 
unnamed_Operator_acceptance 
)}
unnamed_acceptance_response_message));

R2 := if (Stimulus_request_message_bool(unnamed_Operator_search_request,
unnamed_search_request_request_message)) then (
send_response_message_bool((unnamed_Operator_search_response,
unnamed_search_response_response_message)) AND
(the_search_response_is_sorted_alphabetically));

R3 := if (Stimulus_request_message_bool(unnamed_Remote_Operator_search_request,
unnamed_search_request_request_message)) then (
send_response_message_bool((
unnamed_Remote_Operator_search_response,
unnamed_search_response_response_message)) AND
(the_search_response_is_sorted_alphabetically));

R4 := if (Stimulus_request_message_bool(unnamed_VPL_search_request,
unnamed_search_request_request_message)) then (
send_response_message_bool((
unnamed_VPL_search_response, unnamed_search_response_response_message))
AND (the_search_response_is_sorted_alphabetically));

R5 := if (Stimulus_request_message_bool(unnamed_SFU_search_request,
Chapter 4 Stimulus Response Requirements Specification Notation

unamed_search_request_request_message )) then ( send_response_message_bool (( unamed_SFU_search_response , unamed_search_response_response_message)) AND (the_search_response_is_sorted_alphabetically)); R6 := if ( Stimulus_request_message_bool ( unamed_UVIC_search_request, unamed_search_request_request_message )) then ( send_response_message_bool (( unamed_UVIC_search_response , unamed_search_response_response_message)) AND (the_search_response_is_sorted_alphabetically)); R7 := if ( Stimulus_request_message_bool ( unamed_EPL_search_request, unamed_search_request_request_message )) then ( send_response_message_bool (( unamed_EPL_search_response , unamed_search_response_response_message)) AND (the_search_response_is_sorted_alphabetically)); Specification := R1 AND R2 AND R3 AND R4 AND R5 AND R6 AND R7;
4.8.3 Test Specification Generation

The test specifications generated using the TCG [Don98] are listed below.

```
%tcg -D -T 30 30 30 Specification

Spec Specification: Test Frames for R1 AND R2 AND R3 AND R4 AND R5 AND R6 AND R7

Generating test classes...
Generating test classes took 9 seconds.
(7 seconds spent rewriting.)
Rewriting quantifiers for frame stimuli took 1 second.

Generating test frames took 0 seconds.
```

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Condition</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Class 1 (detailed, implicant) antecedent DNF/CNF = 1/2 :</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--Test Frame 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1) Stimulus_request_message_bool(, unnamed_Operator_search_request unnamed_search_request_request_message)</td>
<td>1) return_response_message_bool(, unnamed_Operator_acceptance unnamed_acceptance_response_message)</td>
<td></td>
</tr>
<tr>
<td>2) The_input_is_not_rejected_bool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 4 Stimulus Response Requirements Specification Notation

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Condition</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Class 2 (detailed, implicant) antecedent DNF/CNF = 1/1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--Test Frame 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stimulus</strong></td>
<td><strong>Condition</strong></td>
<td><strong>Criteria</strong></td>
</tr>
<tr>
<td>1) Stimulus_request_message Bool(), named_Operator_search_request named_search_request_message)</td>
<td>1) send_response_message Bool(), named_Operator_search_response named_search_response_message)</td>
<td>2) the search_response_is_sorted_alphabetically</td>
</tr>
</tbody>
</table>

| Test Class 3 (detailed, implicant) antecedent DNF/CNF = 1/1: |
| --Test Frame 1: |
| **Stimulus** | **Condition** | **Criteria** |
| 1) Stimulus_request_message Bool(), named_Operator_search_request named_search_request_message) | 1) send_response_message Bool(), named_Remote_Operator_search_response named_search_response_message) | 2) the search_response_is_sorted_alphabetically |

<p>| Test Class 4 (detailed, implicant) antecedent DNF/CNF = 1/1: |
| --Test Frame 1: |
| <strong>Stimulus</strong> | <strong>Condition</strong> | <strong>Criteria</strong> |
| 1) Stimulus_request_message Bool(), named_Operator_search_request named_search_request_message) | 1) send_response_message Bool(), named_VPL_search_response named_search_response_message) | 2) the search_response_is_sorted_alphabetically |</p>
<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Condition</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Class 5 (detailed, implicant) antecedent DNF/CNF = 1/1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>--Test Frame 1:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1)</td>
<td>1) send_response_message_bool(, unnamed_SFU_search_response_unamed_search_response_message)</td>
</tr>
<tr>
<td></td>
<td>Stimulus_request_message_bool(, unnamed_Operator_search_request_unamed_search_request_message)</td>
<td>2) the search_response_is_sorted_alphabetically</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Condition</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Class 6 (detailed, implicant) antecedent DNF/CNF = 1/1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>--Test Frame 1:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1)</td>
<td>1) send_response_message_bool(, unnamed_UVIC_search_response_unamed_search_response_message)</td>
</tr>
<tr>
<td></td>
<td>Stimulus_request_message_bool(, unnamed_Operator_search_request_unamed_search_request_message)</td>
<td>2) the search_response_is_sorted_alphabetically</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Condition</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Class 7 (detailed, implicant) antecedent DNF/CNF = 1/1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>--Test Frame 1:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1)</td>
<td>1) send_response_message_bool(, unnamed_EPL_search_response_unamed_search_response_message)</td>
</tr>
<tr>
<td></td>
<td>Stimulus_request_message_bool(, unnamed_Operator_search_request_unamed_search_request_message)</td>
<td>2) the search_response_is_sorted_alphabetically</td>
</tr>
</tbody>
</table>
4.8.4 Discussion of the Example

The SRRS requirements specification notation is convenient for describing requirements which have a large number of stimuli/responses that are processed/generated in the same way. For example, the second requirement in the example above represents six requirement statements. Rather than repeating the requirement statement six times, the author can use the notation to remove redundancy in the specification. In the example given, the second requirement is:

2) Upon receipt of a [search request], the library system shall send a [search response]. {The search response is sorted alphabetically}.

The six requirements that are actually represented in the SRRS notation are:

2) Upon receipt of a Operator <search request>, the library system shall send a Operator <search response>. {The search response is sorted alphabetically}.

2) Upon receipt of a “Remote Operator” <search request>, the library system shall send a “Remote Operator” <search response>. {The search response is sorted alphabetically}.

2) Upon receipt of a VPL <search request>, the library system shall send a VPL <search response>. {The search response is sorted alphabetically}.

2) Upon receipt of a SFU <search request>, the library system shall send a SFU <search response>. {The search response is sorted alphabetically}.
2) Upon receipt of a UVIC <search request>, the library system shall send a UVIC <search response>. {The search response is sorted alphabetically}.

2) Upon receipt of a EPL <search request>, the library system shall send a EPL <search response>. {The search response is sorted alphabetically}.

The single requirement statement documented in the SRRS notation is compact. The expansion of the one statement into the six is achieved using the algorithms described in Section 4.5.

4.9 Conclusions

For large systems, the usefulness as a communication tool, maintainability, consistency and conciseness are important characteristics of a requirement specification to consider. The partitioning of the requirement specification provides the customer with an overview of the capabilities of the system, as seen from the end users point of view. A maintainable specification is described using techniques that support a blackbox description of the system. This helps to avoid describing internal processing which may constrain the design or create additional work to interpret the requirements for the system. Writing conventions in a specification technique promote writing the requirements in a consistent style. This is of particular interest when a large number of authors are developing the specification over a lengthy period of time. The writing constraints also promote describing a common idea with the same phrasing and vocabulary in different parts of the specification. To encourage a concise description of the requirements, a mechanism to group inputs and outputs that trigger the same requirements and are generated by the same requirements is useful for systems with a large number of unique input and outputs. This
grouping allows the requirements to be written once, rather than for each member in the group.

SRRS uses an external partitioning of the requirements; promotes a stimulus response (blackbox) style; avoids computer human interface (CHI) details; supports the description of data and their relationships; and has a defined syntax that looks like natural language. These concepts and ideas are used to assist the author in writing understandable, unambiguous, concise, precise, verifiable, and maintainable specifications.

The SRRS notation has an English-like concrete syntax. However, the braces used in the concrete syntax to support the unambiguous parsing of a logical condition or data expression complicate the requirement specification and make it look less like English text. One way to work around this shortcoming of the notation is to provide additional tool support that could either hide the braces (remove them from sight temporarily) or strip the braces from the file (remove them permanently). Removing the braces may introduce errors or ambiguities in logical conditions that are boolean expressions, however, if the braces have been used to over-ride precedence rules.
Chapter 5  Data Specification Notation

5.1  Introduction

The principles of modularity (dividing a complex system into simpler pieces) and separation of concerns (dealing with individual aspects of a problem separately) are fundamental in software engineering [Ghe91]. These principles are applied in this work by dividing the problem of specifying a system with complex data relationships into two subproblems, or modules. The first module is concerned with providing requirements authors with a notation for describing the requirements. The notation defined to support this activity is SRRS (refer to Chapter 4). The second module is concerned with providing users with a means of describing the data in the system; describing the relationships between the data; typechecking data expressions and logical conditions; and conveniently and unambiguously referencing data in data expressions and logical conditions. The data specification (DSPEC) notation is defined to support these activities.

A concrete syntax is defined for the DSPEC notation and is implemented in the tool support for the notation (refer to Chapter 6). The detailed specification for the notation is in Appendix B. The syntax for introducing the context for the DSPEC notation is straightforward. The context for DSPEC is composed of declarations for data objects, data types, data subtypes, associations and functions. This context is used when a data expression or a logical condition is checked to determine if it the data references typecheck correctly and are unambiguous. The rules for determining if a data reference is unambiguous are defined in the DSPEC notation using graph theory.

In this work, the data references are checked using algorithms that are based on graph theory. Each data type in the context is represented by a node in a directed graph. The functions are represented by arcs in a directed graph. An unambiguous data reference is defined by the
presence of exactly one path in the directed graph from one type to another type. The paths are found using a set of rules described in this chapter.

This chapter is organized as follows. Following the introduction, the evolution and important characteristics of the DSPEC notation are presented in Section 5.2. An overview of DSPEC is presented in Section 5.3. The evaluation rules for data references are described in Section 5.4. Data expressions and logical conditions supported in DSPEC are presented in Sections 5.5 and 5.6. The translation of DSPEC into S is in Section 5.7. The conclusions for the chapter are in Section 5.8.

5.2 Evolution and Characteristics of DSPEC

The DSPEC notations is a typed notation that is designed to support the description of complex data expressions and logical conditions. DSPEC is a typed notation and is specified with typechecking capabilities. The significant advantage of DSPEC over other notations that support the description of data and their relationships is that the DSPEC notation supports a mechanism that simplifies referencing data and provides feedback to the user about these data references. The basic idea is that DSPEC can build the relationships, or paths, between data types based on the context. This alleviates the user from having to specify every function application that is needed to reference one type to another type.

For data expressions, the DSPEC notation uses concepts from established data modelling techniques including entity relationship diagrams and data dictionaries. DSPEC can be used to describe data types, data subtypes, data objects, and the relationships between these data entities. Commonly used data types and objects are built into the notation (e.g., units of time, distance, temperature, and volume, bool, TRUE, FALSE). Domain specific data entities, functions, and
predicates may be declared by the user.

For logical conditions, the DSPEC notation supports the basic boolean operators (not, and, or) and predicates. The important characteristics of the notation and where they have evolved from are described in this section.

5.2.1 Type Checks

DSPEC is a typed notation that uses both typechecking and type-inferencing when evaluating data expressions and logical conditions. A typecheck asks the question “are you of type x?” whereas a type-inference asks the question “what type could you be?”. A significant advantage of a typed specification notation over a non-typed notation is the ability to support mechanized, or automated, type checking [Lam98]. Automated support can make the detection of errors in a specification faster to detect and correct.

5.2.2 Building Paths

DSPEC is designed to simplify making data references. When a user makes a data reference, they do not need to specify every function application that is necessary to refer to the data needed. The DSPEC notation is designed to build these function applications, or paths, for the user. The following simple example illustrates the mechanism DSPEC supports to build paths for the user. In this example, there are three data types (<type 0>, <type 1>, and <type 2>) and two functions (f1 and f2) in the context (refer to Figure 5.1). In the figure, the circles represent the type sets and the arcs represent the functions that relate a domain with a co-domain.

If the user needs to reference a data element in <type 2> that is associated with a data element in <type 0>, then the user can describe this evaluation request as follows in DSPEC:
Based on this request, the DSPEC notation builds the function applications needed to go from \(<\text{type 0}\)> to \(<\text{type 2}\)> from the associations and function declarations in the context. DSPEC builds these nested function applications for the user:

\[
f_2 (f_1 \text{ unnamed\_type\_0})
\]

In contrast, a request to evaluate a data reference written in a higher order logic requires that the user explicitly write all the function applications. For example, the same request in S requires the user to write the nested function applications themselves:

\[
f_2 (f_1 \text{ unnamed\_type\_0})
\]

For small examples like this, having the user build the function applications may not be an issue. However, in a large complex data model, DSPEC is designed to simplify referencing data from the user's perspective. DSPEC also allows the user to reference the data by building the function applications, too.

### 5.2.3 Data Expressions

#### 5.2.3.1 Data Dictionary

The idea of using a data dictionary to define the data entities or data types in a domain is an established technique that has been used in structured analysis for over 15 years [Hat88, War85]. The data dictionary provides a centralized repository in which the data for a system development project can be captured. Advantages of a centralized data dictionary include that it requires the requirements authors to document the data they are referring to in the domain in one
place and reduces the introduction of multiple terms that represent the same data (e.g., the terms on-shelf and reshelved). This simplifies the review and interpretation of the requirements specification. A disadvantage of a data dictionary is that it is restricted to describing composition relationships between data types. For example, the data relationship describing that a "Borrower" data type is composed of a number of data types including name, address, outstanding fines, and charged (signed out) items can be described. However, a data dictionary cannot be used to describe other types of relationships, such as the inheritance relationship "is a". Data dictionaries are not well suited to describing subtypes or data objects. To describes these in a data dictionary, the user can place this additional information in a comment. For example, the states of a book in the library system such as ON-SHELF, CHARGED, or MENDING are described as a comment in BNF in this manner:

* the book state values may be ON-SHELF, CHARGED, or MENDING *
5.2.3.2 Entity Relationship Diagrams

Entity relationship diagrams (ERDs) are a graphical notation that can be used to describe relationships between two or more data types. ERDs are also a well established notation that has been used for over 20 years in database development work to describe data types and their relationships. A significant advantage of this notation is that it is extremely flexible. The user may define any kind relationship they choose between two data types (e.g., “is a”, “uses”, “is composed of”) the cardinality of the relationship (e.g., N:M), and the participation constraint (total or partial). In practice, two disadvantages of ERDs have been that they have not been integrated with the functional descriptions of the requirements and that for large projects the ERDs become difficult to work with because there is no encapsulation technique supported.

5.2.4 Logical Conditions

The DSPEC notation supports the boolean operators “not”, “and”, and “or”. These operators may be used with data entities of type bool (data types, data subtypes, data objects) or predicates. For convenience, the type <bool> is built into the DSPEC notation.

5.3 Overview of DSPEC

The DSPEC notation supports the description of the data entities and the relationships between the data entities that needed by the user. A data specification may be written in the DSPEC notation. A data specification is composed of seven parts: data types, data subtypes, data objects, function declarations, associations, data expressions, logical conditions (refer to Figure 5.2). The first five parts are used to build the DSPEC context. The last two parts are used to parse a request to evaluate a data expression or logical condition with respect to the DSPEC context. For example, a data expression is evaluated with respect to the data types, subtypes, objects,
functions and associations declared using a set of rules defined in the notation. A logical condition is evaluated with respect to the boolean operators, the predicates applied, and all of the data referred to in the logical condition. Note that a parameter for a predicate may be a data expression.

The purpose of each of these sections is summarized below. The detailed specification of the notation is presented in Appendix B.

### 5.3.1 Data Types

The purpose of this section is to introduce the user declared data types into the DSPEC context. A data type is a set of data objects. For example, the set of integers is a data type that includes the data objects {..., -3, -2, -1, 0, 1, 2, 3, ...}. In the DSPEC syntax, the declaration of the
Chapter 5 Data Specification Notation

data type integer is:

1):<integer>.

5.3.2 Data Subtypes

The purpose of this section is to introduce the user declared subtypes into the DSPEC context. A data subtype is a subset of a base type. For example, the set of positive integers is a subtype of the base type integer. In the DSPEC syntax, the declaration of the subtype positive integers is:

1) <positive integer> is a subtype of <integer>.

When the user declares a subtype, the DSPEC notation automatically introduces a function that relates the base type to the subtype. The parameters are included as part of the function body to ensure the function is unique. The function introduced for the subtype above is:

1) "<positive integer> is a subtype of <integer>" (<integer>) : <positive integer>.

5.3.3 Data Objects

The purpose of this section is to introduce the user declared data objects. A data object is the name of a unique entity of a specific type. For example, the data object 42 is a unique entity in the integer data type. The declaration of 42 is:

1) 42: <integer>.
5.3.4 Association Declarations

The purpose of this section is to introduce the user declared associations into the DSPEC context. An association is a simple function declaration. The cardinality and the participation constraints are defined in the declaration. The parameters are delimited with parentheses in the function declaration. An example of an association declaration is:

1) Every <type 0> is associated with exactly one <type 1>.

5.3.4.1 Cardinality and Participation of Associations

In the previous example, the association that has been used describes a 1:1 cardinality relationship with total participation constraints. The DSPEC notation supports the description of more general cardinality relationships and both partial and total participation constraints to provide the flexibility necessary in large systems with complex data relationships. The cardinality describes the number of elements that may be involved in the relationship (e.g., N:M,1:M,N:1). The associations that use cardinality N or M in the relationship are referring to a set of data elements. The associations that use cardinality 1 in the relationship are referring to an individual data element. The participation constraint describes whether a data element is required to participate (total participation) or is optional (partial participation). It is important to note that when an association with partial participation is used by the notation to build a path, the assumption used is that the data element is participating. The associations supported in DSPEC are summarized in Table 5.1.

5.3.5 Function Declarations

The purpose of this section is to introduce the user declared functions and predicates into
<table>
<thead>
<tr>
<th>Cardinality</th>
<th>Participation Constraint</th>
<th>DSPEC Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>type 1: total</td>
<td>1 &lt;type 1&gt; is associated with 1 &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: total</td>
<td></td>
</tr>
<tr>
<td>1:M</td>
<td>type 1: total</td>
<td>1 &lt;type 1&gt; is associated with 1 or more &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: total</td>
<td></td>
</tr>
<tr>
<td>1:M</td>
<td>type 1: total</td>
<td>1 &lt;type 1&gt; is associated with 0 or more &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: partial</td>
<td></td>
</tr>
<tr>
<td>N:1</td>
<td>type 1: total</td>
<td>1 or more &lt;type 1&gt; is associated with 1 &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: total</td>
<td></td>
</tr>
<tr>
<td>N:1</td>
<td>type 1: total</td>
<td>1 or more &lt;type 1&gt; is associated with 1 or more &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: partial</td>
<td></td>
</tr>
<tr>
<td>N:M</td>
<td>type 1: total</td>
<td>1 or more &lt;type 1&gt; is associated with 0 or more &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: partial</td>
<td></td>
</tr>
<tr>
<td>N:M</td>
<td>type 1: partial</td>
<td>0 or more &lt;type 1&gt; is associated with 1 or more &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: total</td>
<td></td>
</tr>
<tr>
<td>N:M</td>
<td>type 1: partial</td>
<td>0 or more &lt;type 1&gt; is associated with 0 or more &lt;type 2&gt;</td>
</tr>
<tr>
<td></td>
<td>type 2: partial</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1 Cardinallity and Participation Constraints

the DSPEC context. Note that a predicate is a special case of a function, as its return type is always bool. A user declared function or predicate has total participation. A function declaration is written in the “flex-fix” notation. This format does not restrict the user to writing the functions in pre-fix, post-fix, or in-fix. Instead, the user can intersperse the parameters in the body of the function as they need to. The parameters are delimited with parenthesis in the function declaration. This allows the function declaration to be structured more like an English sentence. An example of a predicate declaration is:

1) “The x coordinate is less than” (<integer>) “and greater than” (<integer>): bool.
5.3.6 Data Expressions

The data expressions in this section may be evaluated in two ways. The first is evaluating the type of the data expression using type-checking and type-inference rules. The second is determining if a data expression references the data in the context unambiguously or not. The rules defined in the DSPEC notation are used to do the evaluation (refer to Section 5.4). The data expressions supported in DSPEC are described in detail in Section 5.5.

The simplest data expressions include the request to evaluate a data type, data subtype, or data object. In these cases, the data entity is looked up in the context. If the data entity is found, then the evaluation type returned to the user is the base type of the data entity and the evaluation is considered successful. An example of this kind of data expression is:

1) 42.

Data expressions also support the request to evaluate a function application. The parameters of a function application are data expressions. Each parameter's data expression is evaluated according to the data expression evaluation rules. An example of a function application is:

2) The product of {42} and {10}.

In this example, the parameters 42 and 10 are simple data references. More complex applications may have functions as their parameters. If this is the case, each parameter of each function application is checked in the evaluation.

The data expression section may also be used to determine if there is an unambiguous data reference. The user may request to determine if there is a unique path from one data entity to
another data entity, based on the context. An example of such a request is:

1) \{\text{the } \{<\text{type 1}>\} \text{ associated with } \{<\text{type 0}>\}\}\}

The response to this request is a list of all possible paths found in the directed graph that begin at the node representing \(<\text{type 0}>\) and end at the node representing \(<\text{type 1}>\).

5.3.7 Logical Conditions

The request to evaluate a logical condition is a special case of evaluating a data expression. The logical conditions may be references to data entities of type bool, predicate applications, or may be combined using boolean operators. The logical conditions must evaluate to type bool in order to pass the evaluation. The logical conditions supported in DSPEC are described in detail in Section 5.6.

The simplest logical conditions include the request to evaluate a data type, data subtype, or data object. In these cases, the base type of the data entity is looked up in the context. If the base type is \(<\text{bool}>\), then the data entity passes the evaluation check. An example of this logical condition is:

1) \{\text{TRUE}\}.

Logical conditions also support the request to evaluate a predicate application. The parameters of a predicate application are data expressions. Each data expression is evaluated according to the data expression evaluation rules. An example of a predicate application is:

2) \{\text{The } x \text{ coordinate is less than } \{42\} \text{ and greater than } \{10\}\}. 

104
In this example, the parameters 42 and 10 are simple data references. More complex predicate applications may have functions as their parameters.

Boolean operators (not, and, or) are available to the user, allowing them to build more complicated logical conditions. When these operators are used, each of the operands is evaluated. An example of a boolean expression is:

\[ 3) \{\text{TRUE}\} \text{ or } \{\text{The x coordinate is less than } 42 \text{ and greater than } 10\} \].

5.4 Evaluation Rules

The set of rules used to determine if a data reference is unambiguous is described in this section using examples and pseudocode. There are two rules defined: 1) parent rule; 2) building paths rule. These rules are applied when a request is made that requires DSPEC to build all possible paths from one data type to another data type. The rules are used in the order listed above. For example, the first evaluation is done using the parent rule. If this is successful, then the evaluation for the data reference succeeds. If the parent rule evaluation is not successful, then the build paths rule is applied in the evaluation. If the evaluation using the building paths rule is successful, then the evaluation is successful. Otherwise, the evaluation fails.

A small, DSPEC context is used in this section to provide a working example. The context is composed of:

Data Types:

1) :<this type>.
Chapter 5 Data Specification Notation

2) :<that type>.

3) :<the other type>.

4) :<some type>.

5) :<some other type>.

6) :<integer>.

Data Objects:

1) important_constant:<integer>.

2) important_constant:<some other type>.

3) extremely_important_constant:<this type>.

Function Declarations:

1) "my function body"(<this type>) : <that type>.

2) "my other function body"(<this type>) : <that type>.

3) "another function body" (<the other type>) : <this type>.

4) "some function body" (<this type>) : <the other type>.

5) "a function body" (<the other type>) : <some type>.

6) "this function body" (<some type>) : <some other type>.

The directed graph of the context is derived from these declarations (refer to Figure 5.3).

The nodes in the graph represent the data types. The arcs in the graph represent the functions.
5.4.1 Parent Rule

The parent rule is used to determine if there is exactly one parent child relationship in the directed graph representing the context. If the parent rule applies, the data reference is unambiguous. For example, the parent rule applies to the data references summarized in Table 5.2.

<table>
<thead>
<tr>
<th>From Data Reference</th>
<th>To Data Reference</th>
<th>Parent Rule Applies</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.2 Example of Parent Rule Evaluations
Table 5.2 Example of Parent Rule Evaluations

<table>
<thead>
<tr>
<th>From Data Reference</th>
<th>To Data Reference</th>
<th>Parent Rule Applies</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;that type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;some type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;some other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;integer&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;this type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;some type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;some other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;that type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;integer&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;some other type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;this type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;that type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;integer&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;some type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>any type</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some other type&gt;</td>
<td>any type</td>
<td>No</td>
</tr>
<tr>
<td>&lt;that type&gt;</td>
<td>any type</td>
<td>No</td>
</tr>
</tbody>
</table>

The pseudocode for the parent rule is:

get from type

get to type
initialize number of paths = 0

for every function declaration

    if ((domain = from type) and (range = to type))

        increment number of paths

    return number of paths

If the number of paths returned is equal to one, then the parent rule applies and the data reference is unambiguous. If the number of paths returned is greater than one, then the parent rule does not apply and the data reference is ambiguous. If the number of paths is zero, then the parent rule does not apply. In this case, the data reference is not known to be ambiguous or unambiguous. Another rule, the building paths rule, needs to be applied to determine if there is unique path in the graph.

5.4.2 Building Paths Rule

The building paths rule uses a depth first search to determine the paths in the graph from the from data type to the to data type. The search starts at the from type. Every possible path is attempted to reach the to data type in the data reference. For example, if the from data type is <the other type> and the to type is <some other type>, the following path is found in the graph (refer to Figure 5.3):

path 1

    from <the other type> to <some type>

    from <some type> to <some other type>
Since exactly one path is found, this data reference is unambiguous.

The graph is not restricted to being an acyclic graph and an additional check must be used in the search to determine if a path has a cycle in it. If this check is not made, then the data reference may have an infinite number of paths and the search does not terminate. If a cycle is detected, then the path currently being built is added to the paths and the next possible path is attempted. For example, if the from data type is <this type> and the to data type is <some other type>, then the following paths are found in the graph:

path 1

from <this type> to <the other type>

from <the other type> to <some type>

from <some type> to <some other type>

path 2

from <this type> to <the other type>

from <the other type> to <this type>

from <this type> to <the other type>

from <the other type> to <some type>

from <some type> to <some other type>

Since more than one path is found, this data reference from <this type> to <some other
type > is ambiguous. In this example, the loop in the graph is detected when the second path is built.

5.5 Data Expressions

One of the characteristics of the DSPEC notation is the flexibility in describing and referring to data and their relationships. The notation supports data expressions that allow the user to reference individual data entities, associations between data entities, and use functions. Using an example context, the evaluation type and the interpretation of the expressions are illustrated the following sections.

5.5.1 Simple Data Entity

The simplest data expression in the DSPEC notation is a reference to a single data entity in the context. The data entity may be a data type, data subtype, or a data object. When this expression is used, the evaluation type of the expression is the data type of the entity. The interpretation of this data expression depends on the entity that is used. The data expression allows the user to reference an un-named data object of a particular type or subtype, or a named data object. To illustrate this data expression, a symbolic example is provided here. An example context is initially composed of the following data types, subtypes, and data objects:

Data Types:

:<type 0>

Subtypes:

<subtype 0>:<type 0>

Data Objects:
Chapter 5 Data Specification Notation

My_constant :<type 0>

The DSPEC notation supports the data references in Table 5.3 using this data expression that references a simple data entity. Rows one and two of the table provide an example of the data expression using a data type and a data subtype entity. In these cases, when the user makes a reference to a data type or a subtype, the interpretation is that the user is referring to one of the elements in the type set, but the user does not explicitly state which element. The third row provides an example of using a data object as the data entity. In this case, the data element referenced in the type set is clearly identified by the user. All the data references in this example context typecheck successfully.

<table>
<thead>
<tr>
<th>ID #</th>
<th>Data Expression</th>
<th>Data Expression Evaluation Type</th>
<th>Interpretation of Data Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ &lt;type 0&gt; }</td>
<td>&lt;type 0&gt;</td>
<td>The reference is to an un-named object of type &lt;type 0&gt;</td>
</tr>
<tr>
<td>2</td>
<td>{ &lt;subtype 0&gt; }</td>
<td>&lt;subtype 0&gt;</td>
<td>The reference is to an un-named object of subtype &lt;subtype 0&gt;</td>
</tr>
<tr>
<td>3</td>
<td>{ My_constant }</td>
<td>&lt;type 0&gt;</td>
<td>The reference is to the data object My_constant of type &lt;type 0&gt;</td>
</tr>
</tbody>
</table>

Table 5.3 Data Expression: Example of Basic Data Entity Reference

5.5.2 Association Between Two Data Entities

A more complex data expression is used to reference one data entity that is associated, or related, either directly or indirectly to a second data entity. The data entities may be a data type, data subtype, data object, or function application. The type evaluation of this expression is the type of the first data entity. In order to illustrate this data expression, an example context is composed of the following constants, types, subtypes, and associations:

Data Types:
Subtypes:

<subtype 0>:<type 0>

<subtype 1>:<type 1>

Data Objects:

My_constant :<type 0>

Associations:

1 {<type 0>} is associated with 1 {<type 1>}

1 {<type 1>} is associated with 1 {<type 2>}

The association references the user may request for evaluation based on this context are summarized in Table 5.4.

<table>
<thead>
<tr>
<th>ID #</th>
<th>Data Expression</th>
<th>Data Expression Evaluation Type</th>
<th>Interpretation of Data Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{the {&lt;type 1&gt;} associated with {&lt;type 0&gt;}}</td>
<td>&lt;type 1&gt;</td>
<td>The reference is to an un-named data object of type &lt;type 1&gt; that is associated with an un-named data object of type &lt;type 0&gt;</td>
</tr>
<tr>
<td>2</td>
<td>{the {&lt;type 1&gt;} associated with {My_constant}}</td>
<td>&lt;type 1&gt;</td>
<td>The reference is to an un-named data object of type &lt;type 1&gt; that is associated with a named data object My_constant of type &lt;type 0&gt;</td>
</tr>
</tbody>
</table>

Table 5.4 Data Expression: Examples of Association References
Chapter 5 Data Specification Notation

<table>
<thead>
<tr>
<th>ID #</th>
<th>Data Expression</th>
<th>Data Expression Evaluation Type</th>
<th>Interpretation of Data Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>{the {&lt;type 2&gt;} associated with {&lt;type 1&gt;}}</td>
<td>&lt;type 2&gt;</td>
<td>The reference is to an un-named data object of type &lt;type 2&gt; that is associated with an un-named data object of type &lt;type 1&gt;</td>
</tr>
<tr>
<td>4</td>
<td>{ the [&lt;subtype 1&gt;] associated with [&lt;type 0&gt;]}</td>
<td>&lt;subtype 1&gt;</td>
<td>The reference is to an un-named data object of subtype &lt;subtype 1&gt; that is associated with an un-named data object of type &lt;type 0&gt;</td>
</tr>
<tr>
<td>5</td>
<td>{ the [&lt;subtype 1&gt;] associated with [My_constant]}</td>
<td>&lt;subtype 1&gt;</td>
<td>The reference is to an un-named data object of subtype &lt;subtype 1&gt; that is associated with a named data object My_constant of type &lt;type 0&gt;</td>
</tr>
<tr>
<td>6</td>
<td>{the {&lt;type 2&gt;} associated with {&lt;subtype 1&gt;}}</td>
<td>&lt;type 2&gt;</td>
<td>The reference is to an un-named data object of type &lt;type 2&gt; that is associated with an un-named data object of subtype &lt;subtype 1&gt;</td>
</tr>
<tr>
<td>7</td>
<td>{the {&lt;type 2&gt;} associated with {&lt;type 0&gt;}}</td>
<td>&lt;type 2&gt;</td>
<td>The reference is to an un-named data object of type &lt;type 2&gt; that is associated with an un-named object of type &lt;type 0&gt;</td>
</tr>
<tr>
<td>8</td>
<td>{the {&lt;type 2&gt;} associated with {My_constant}}</td>
<td>&lt;type 2&gt;</td>
<td>The reference is to an un-named data object of type &lt;type 2&gt; that is associated with a named data object My_constant of type &lt;type 0&gt;</td>
</tr>
</tbody>
</table>

Table 5.4 Data Expression: Examples of Association References

The first three rows of the table describe data reference associations between two entity types that have evaluated successfully based on the parent rule. Rows four to eight are examples of *indirect* associations. In these indirect data references, more than one association in the context is used to establish the association between the data entities. These data references are unambiguous based on this context because they have succeeded using the build paths rule.

The association reference in DSPEC is meant to be a general purpose, built-in mechanism
to support describing that a relationship exists between two data entities. If this is not suitable for
the users (e.g., the user would like to have a more specifically named function such as “is a”),
then the users can define their own functions and predicates to relate data entities.

5.5.3 Function Application

The first step in this evaluation is to determine if a matching function declaration exists in
the context. If it does, then each of the parameters in the application is evaluated. If the parameter
evaluations are successful, then the evaluation of the function application is successful.

5.6 Logical Conditions

DSPEC supports the basic boolean operators (not, and, or), user declared data entities of
type bool, and user declared predicates. For the user’s convenience, the basic data entities for
logical conditions including the data type bool and the data objects true and false, are built into
DSPEC. The logical conditions are summarized in Table 5.5.

<table>
<thead>
<tr>
<th>ID</th>
<th>Logical Condition</th>
<th>Logical Condition Evaluation Type</th>
<th>Interpretation of Logical Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{&lt;bool&gt;}</td>
<td>&lt;bool&gt;</td>
<td>The reference is to an un-named object of type &lt;bool&gt;</td>
</tr>
<tr>
<td>2</td>
<td>{ &lt;subtype bool&gt;}</td>
<td>&lt;bool&gt;</td>
<td>The reference is to an un-named object of subtype &lt;subtype bool&gt; with base type &lt;bool&gt;</td>
</tr>
<tr>
<td>3</td>
<td>{ TRUE }</td>
<td>&lt;bool&gt;</td>
<td>The reference is to a named object, TRUE, of type &lt;bool&gt;</td>
</tr>
<tr>
<td>4</td>
<td>The value is less than {42} and greater than {10}</td>
<td>&lt;bool&gt;</td>
<td>The reference is a predicate application that returns an un-named data object of type &lt;bool&gt;. The first parameter is a simple data expression that evaluates to type &lt;integer&gt;. The second parameter is a simple data expression that evaluates to type &lt;integer&gt;.</td>
</tr>
<tr>
<td>5</td>
<td>{{&lt;bool&gt;} or {TRUE}}</td>
<td>&lt;bool&gt;</td>
<td>The reference is a logical or condition that returns an un-named data object of type &lt;bool&gt;.</td>
</tr>
</tbody>
</table>
The logical conditions evaluated in rows 1-3 are typechecked. If the data entities are declared in the context, the evaluation is successful. Row 4 is an example that requests the evaluation of a predicate application. The first step in this evaluation is to determine if a matching predicate declaration exists in the context. If it does, then each of the parameters in the application is evaluated. If the parameter evaluations are successful, then the evaluation of the logical condition has a successful evaluation. Rows 5-7 are examples of requests to evaluate boolean expressions. DSPEC supports the “not”, “and”, and “or” boolean operators. Each of the operands is evaluated using the appropriate rules. If the operands evaluate successfully, then the logical condition has a successful evaluation.

The DSPEC notation also supports existentially and universally quantified expressions. The quantified variable may be either a data type or a data subtype in a logical condition. These expressions are summarized in Table 5.6.


### Quantified Expressions

<table>
<thead>
<tr>
<th>Quantified Expression</th>
<th>Quantified Expression Evaluation Type</th>
<th>Interpretation of Quantified Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ one or more of the &lt;subtype 0&gt; such that { the colour of {&lt;subtype 0&gt;} is blue} }</td>
<td>&lt;bool&gt;</td>
<td>The reference is to an existential quantifier for the subtype &lt;subtype 0&gt; in the predicate application.</td>
</tr>
<tr>
<td>{ all of the &lt;type 1&gt; such that {the colour of {&lt;type 1&gt;} is green} }</td>
<td>&lt;bool&gt;</td>
<td>The reference is to a universal quantifier for the type &lt;type 1&gt; in the predicate application.</td>
</tr>
<tr>
<td>{ all of the &lt;subtype 1&gt; such that {the colour of {&lt;subtype 1&gt;} is green} }</td>
<td>&lt;bool&gt;</td>
<td>The reference is to a universal quantifier for the subtype &lt;subtype 1&gt; in the predicate application.</td>
</tr>
<tr>
<td>{ none of the &lt;type 1&gt; such that {the colour of {&lt;type 2&gt;} is orange} }</td>
<td>&lt;bool&gt;</td>
<td>The reference to the negation of an existential quantifier for the type &lt;type 2&gt; in the predicate application.</td>
</tr>
<tr>
<td>{ none of the &lt;subtype 2&gt; such that {the colour of {&lt;subtype 2&gt;} is orange} }</td>
<td>&lt;bool&gt;</td>
<td>The reference is to the negation of an existential quantifier for the subtype &lt;subtype 2&gt; in the predicate application.</td>
</tr>
</tbody>
</table>

### Table 5.6 Quantified Expressions

#### 5.7 Translation into S

Translating DSPEC into S, a higher order logic, is used to provide an operational semantics for the DSPEC notation. The translations of the DSPEC data expressions are summarized in Table 5.7. Examples of translations are provided in this section to demonstrate the translation.

<table>
<thead>
<tr>
<th>Data Specification Component</th>
<th>S Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Name Declaration</td>
<td>Type Declaration</td>
</tr>
<tr>
<td>Subtype Declaration</td>
<td>Type Declaration (unique name guaranteed)</td>
</tr>
<tr>
<td></td>
<td>Constant Declaration (function, unique name guaranteed)</td>
</tr>
<tr>
<td></td>
<td>Constant Declaration (un-named data object)</td>
</tr>
<tr>
<td>Data Object Declaration</td>
<td>Constant Declaration (unique name guaranteed)</td>
</tr>
<tr>
<td>Association Declaration</td>
<td>Constant Declaration (unique name guaranteed)</td>
</tr>
<tr>
<td>Function Declaration</td>
<td>Constant Declaration</td>
</tr>
</tbody>
</table>

Table 5.7 Summary of Translating DSPEC Components Into S
Chapter 5 Data Specification Notation

### Table 5.7 Summary of Translating DSPEC Components Into S

<table>
<thead>
<tr>
<th>Data Specification Component</th>
<th>S Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate Declaration</td>
<td>Constant Declaration</td>
</tr>
<tr>
<td>Data Expression</td>
<td>Function Application or Data Object</td>
</tr>
<tr>
<td>Logical Condition</td>
<td>Predicate Application or Boolean Expression</td>
</tr>
</tbody>
</table>

The S notation has a number of rules that must be adhered to if the translation is to successfully typecheck with the S tool support, FUSS. These rules are provided in the list below.

1. White space is used as a delimiter in S. In the translation from DSPEC to S, whitespace in the identifier for a data type, for example, is replaced by an underscore.

2. S does not support the flex-fix notation. The function and predicate declarations and applications must be converted from flex-fix into pre-fix notation.

3. S does not support the declaration of subtypes. The subtypes are translated into S by declaring a type name for the subtype and declaring a function that related the subtype to the base type.

4. Every parameter in a function application must be bound to a constant in S. DSPEC allows the user to reference a data type, subtype, or object in a data expression. In S, however, only data objects are allowed. This means that the data types and subtypes must be bound when translating a function application into S. For example, a reference to the data type `<type 1>` in a DSPEC function application is bound in the translation to the data object called `unnamed_type_1` with base type `<type_1>`.

5. The logical conditions are only recognized in S when written in upper case (i.e., “OR”, “AND”, and “NOT”).

6. Integer constants do not need to be declared as data objects in S. For example, 42 is declared with the syntax:
S uses the identifier "num" to represent integers.

7. S cannot create paths that relate one data type to another. FUSS can only typecheck the function applications that are explicitly provided. This means that for each indirect data reference the path must be created (by DSPEC or the author) and then translated into S.

### 5.7.1 Example Data Context Translations

An example of translating the DSPEC data type, data subtype, data object, association, and function declarations is provided in Table 5.8.

<table>
<thead>
<tr>
<th>DSPEC</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type</td>
<td>declarations in S, <code>:&lt;type_1&gt;;</code></td>
</tr>
<tr>
<td></td>
<td><code>unamed_type_1:type_1;</code></td>
</tr>
<tr>
<td>Data Subtype</td>
<td>declarations in S, <code>:&lt;subtype_1&gt;;</code></td>
</tr>
<tr>
<td></td>
<td><code>:&lt;subtype_1&gt;_is_a_subtype_of_&lt;type_1&gt; : &lt;type_1&gt; -&gt; &lt;subtype_1&gt;;</code></td>
</tr>
<tr>
<td></td>
<td><code>unamed_subtype_1:subtype_1;</code></td>
</tr>
<tr>
<td>Data Object</td>
<td>declarations in S, <code>:&lt;type_1&gt;;</code></td>
</tr>
<tr>
<td></td>
<td><code>My_constant:&lt;type_1&gt;;</code></td>
</tr>
<tr>
<td>Association</td>
<td>declarations in S, <code>exactly_1_&lt;type_1&gt;_is_associated_with_exactly_1_&lt;type_0&gt; : &lt;type_0&gt; -&gt; &lt;type_1&gt;;</code></td>
</tr>
<tr>
<td>Function Declaration</td>
<td><code>The_value_is_less_than_&lt;integer&gt;_and_greater_than_&lt;integer&gt; : &lt;integer&gt; -&gt; &lt;integer&gt; -&gt; &lt;bool&gt;;</code></td>
</tr>
</tbody>
</table>

Table 5.8 Example Translation of Declarations in DSPEC into S
5.7.2 Example Data Expression Translations

Examples of the translation of data expressions written in DSPEC into S are provided in Table 5.9.

<table>
<thead>
<tr>
<th>DSPEC</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple data reference</td>
<td>unnamed_num:num;</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td></td>
</tr>
<tr>
<td>simple data reference</td>
<td>unnamed_positive_num;&lt;positive num&gt;;</td>
</tr>
<tr>
<td>&lt;positive integer&gt;</td>
<td></td>
</tr>
<tr>
<td>simple data reference</td>
<td>42</td>
</tr>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td>association</td>
<td>The_&lt;type_0&gt;<em>associated_with</em>&lt;type_2&gt; (type_2 associated_with_type_1 unnamed_type_1)</td>
</tr>
<tr>
<td>function application</td>
<td>The_value_is_less_than_&lt;num&gt;<em>and_greater_than</em>&lt;num&gt; 42 10;</td>
</tr>
</tbody>
</table>

Table 5.9 Example Translation of Data Expressions in DSPEC into S

a. Path build by DSPEC to relate <type 2> to <type 1>, and then <type 1> to <type 0>.

5.7.3 Example Logical Condition Translations

Examples of translations of logical conditions written in DSPEC into S are provided in Table 5.10.

<table>
<thead>
<tr>
<th>DSPEC</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>NOT unnamed_subtype_bool;</td>
</tr>
<tr>
<td>{not {&lt;subtype bool&gt;}}</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>TRUE AND unnamed_bool;</td>
</tr>
<tr>
<td>{ [TRUE] and {&lt;bool&gt;} }</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>The_value_is_less_than_&lt;num&gt;<em>and_greater_than</em>&lt;num&gt; 42 10 OR FALSE;</td>
</tr>
<tr>
<td>{ {The value is less than {42} and greater than {10}} OR {FALSE}}</td>
<td></td>
</tr>
<tr>
<td>existential quantification</td>
<td>exists (named_type_0: named_type_0).the_&lt;type_0&gt;_is_coloured_blue named_type_0;</td>
</tr>
<tr>
<td>{ one or more of the &lt;type 0&gt; such that {the &lt;type 0&gt; is coloured blue}}</td>
<td></td>
</tr>
<tr>
<td>universal quantification</td>
<td>forall (named_type_1:named_type_1).the_&lt;subtype_1&gt;_is_coloured_green named_subtype_1;</td>
</tr>
<tr>
<td>{ all of the &lt;type 1&gt; such that {the &lt;subtype 1&gt; is coloured green}}</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5 Data Specification Notation

Table 5.10 Translation of Logical Conditions in DSPEC into S

<table>
<thead>
<tr>
<th>DSPEC</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation of universal quantification</td>
<td>NOT (exists (unnamed_subtype_2:subtype_2). the_&lt;subtype_2&gt;_is_coloured_orange unamed_subtype_2; );</td>
</tr>
</tbody>
</table>

5.8 Conclusions

The DSPEC notation is a typed notation that supports the description of data entities (data types, data subtypes, and data objects), the relationships between these data entities (functions and predicates), a simplified means of referencing data, and requests to evaluate data expressions and logical conditions.

The DSPEC notation evaluates these data expressions and logical conditions at two levels. First, the requests are type checked with respect to the context. This provides the user with feedback regarding type mismatches and missing declarations. The user can quickly detect and correct these type errors in a specification. The second evaluation determines whether the data references are unambiguous or not. In a complex data model, data entities may be used in multiple places and referencing the correct one may not be straightforward. The evaluation checks defined in DSPEC can provide the user with immediate feedback and assist them in developing an unambiguous data model. Tool support has been developed to support the evaluations checks in DSPEC and is described in Chapter 6.

The complexity of defining the DSPEC capabilities has warranted the use of the divide and conquer strategy in this work. With this strategy, the DSPEC notation is also available for use by other notations (in addition to the SRRS notation). For example, users wishing to develop a test specification notation could re-use the DSPEC notation as a core, or foundation notation.
Chapter 5 Data Specification Notation

(refer to Figure 5.4). This test specification notation may be useful for users who need to develop test specifications for testing third party software, for example, but do not believe writing a requirements specification would add value to their development programme.

Figure 5.4 Re-use of DSPEC as a Foundation Notation
Chapter 6  Tool Support for SRRS and DSPEC

6.1  Introduction

Tool support has been developed for the SRRS and DSPEC notations. The tool support is needed to demonstrate that the notations are feasible to implement and to support the evaluation of SRRS (refer to Chapter 7 for a discussion of the experimental evaluation). This chapter provides a description of the tool built for the SRRS and the DSPEC notations including the requirements in Section 6.2, architecture in Section 6.3, and a summary of the implementation effort in Section 6.4. Future enhancements are in Section 6.5 and the conclusions for the chapter are presented in Section 6.6. Due to their lengths, the functional requirements for the SRRS notation and the DSPEC notations are in Appendices A and B.

6.2  Software Requirements

The functional requirements for the SRRS notation is available in Appendix A and the functional requirements for the DSPEC notation is available in Appendix B. The non-functional requirements for the tool support are listed below:

1. The input to the tool shall be in an ASCII file.
2. The error messages displayed to the user shall be ASCII text.
3. The implementation language shall be the C programming language.
4. The unix programming tools lex and yacc shall be used to scan and parse the specification.
5. The tool shall be developed for the UNIX platform.
6. The S translation, if requested, shall be output to an ASCII file.
6.3 Architecture Model

The architecture combines a pipeline and layer architectural styles. The pipeline component of the architecture is based on the scanning, parsing, validation, and translation options specified in the SRRS tool. If the user chooses the option to parse the specification, then the scanning and parsing modules are used. Alternatively, if the user chooses to validate the specification, the scanning, parsing, and validating modules are used. Finally, if the user chooses to translate the specification into S, then the scanning, parsing, validating, and translating modules are used.

The layered component of the architecture is the re-use of the DSPEC engine to check data references. When a validation module is used and a logical condition is present, the DSPEC engine is used to determine if the data references in the logical condition are unique. The purpose of using a layered architecture is to support the re-use of the DSPEC engine with other notations. For example, if a notation for describing test cases is defined, the DSPEC engine could be re-used in its tool development.
The architecture is modelled using a structured analysis technique [War85]. The diagrams are in Figures 6.1-6.5. The data and control flow diagrams are combined into one set of diagrams because they are straightforward to model. The process specifications, control specifications, and data dictionary are documented after the figures.

6.3.1 Structured Analysis Diagrams
Figure 6.2 Diagram 1 SRRS System
Figure 6.3 Diagram 1.1 Parse Specification
Figure 6.4 Diagram 1.2 Validate Specification
6.3.2 Primitive Specifications

The process and control specifications are described in this section. They are listed in alphabetical order by name.

Name: Control Parsing, number 1.1.1

Type (process or control): control
Chapter 6 Tool Support for SRRS and DSPEC

Inputs: Parsing request

Outputs: Parsing ok

Description: The Control Parsing CSPEC is a sequential state transition diagram (refer to Figure 6.6).

![State Transition Diagram for CSPEC 1.1.1]

Name: Control Translation, number 1.3.1

Type (process or control): control

Inputs: translation request, validation ok

Outputs: translation ok

Description: The Control Translation CSPEC is a state transition diagram (refer to Figure 6.7).
Figure 6.7 State Transition Diagram for CSPEC 1.3.1

Name: Control Validation, number 1.2.1

Type (process or control): control

Inputs: validation request

Outputs: validation ok

Description: The Control Validation CSPEC is a sequential state transition diagram (refer to
Chapter 6 Tool Support for SRRS and DSPEC

Figure 6.8).
Chapter 6 Tool Support for SRRS and DSPEC

Name: Create Expanded Requirements, number 1.2.4

Type (process or control): process

Inputs: SRRS context, SRRS requirements

Outputs: expanded requirements, validation response, expanded requirements ok

Description: The purpose of the Create Expanded Requirement process is to convert the requirements written in the compact SRRS notation into the stimulus response requirements represented. If an error occurs in the conversion, then an error message is displayed to the user (validation response), the control flow expanded requirements ok is set to False, and conversion ends. Otherwise, the control flow expanded requirements ok is set to True.

Name: Create DSPEC Context, number 1.2.3

Type (process or control): process

Inputs: SRRS context

Outputs: DSPEC context, validation response, DSPEC context ok

Description: The purpose of the Create DSPEC Context process is to convert the SRRS context parse tree into the DSPEC context parse tree. If an error occurs in the conversion, an error message is displayed to the user (validation response), the control flow DSPEC context ok is set to False, and the conversion ends. Otherwise, the control flow DSPEC context ok is set to True.
Name: Create Parse Tree of SRRS Requirements, number 1.1.3

Type (process or control): process

Inputs: token stream

Outputs: parse tree of SRRS requirements, parsing response, parsing ok

Description: The purpose of the Create parse tree of SRRS requirements is to parse the tokens and create a parse tree of the requirements. If an error occurs in the parsing, an error message is displayed to the user (parsing response), the control flow parsing ok is set to False, and parsing ends. Otherwise, the control flow parsing ok is set to True. The unix programming tool yacc is used to describe the grammar rules.

Name: Create Parse Tree of SRRS Context, number 1.1.4

Type (process or control): process

Inputs: token stream

Outputs: parse tree of SRRS context, parsing response, parsing ok

Description: The purpose of the Create parse tree of SRRS context is to parse the tokens and create a parse tree of the context. The context is composed of data objects and data types. If an error occurs in the parsing, then an error message is displayed to the user (parsing response), the
control flow parsing ok is set to False, and parsing ends. Otherwise, the control flow parsing ok is set to True. The unix programming tool yacc is used to describe the grammar rules.

Name: Perform SRRS Validation Checks, number 1.2.2

Type (process or control): process

Inputs: SRRS context, SRRS requirements

Outputs: validation response, SRRS validation ok

Description: The purpose of the process is to perform the validation checks specified in the SRRS notation. If an error is detected, then an error message is displayed to the user (validation response), the control flow SRRS validation ok is set to False, and processing ends. Otherwise, the control flow SRRS validation ok is set to True.

Name: Perform DSPEC Check, number 1.2.5

Type (process or control): process

Inputs: Expanded requirements, DSPEC context

Outputs: validation response, DSPEC check ok

Description: The purpose of the Perform DSPEC Check process is to perform the validation
checks specified in the DSPEC notation. If an error is detected, then an error message is displayed to the user (validation response), the control flow DSPEC check ok is set to False, and processing ends. Otherwise, the control flow DSPEC check ok is set to True.

Name: Translate Specification to S, number 1.3.2
Type (process or control): process
Inputs: SRRS context, Expanded requirements
Outputs: S translation, Translation ok
Description: The purpose of the Translate Specification to S is to translate the expanded requirements and the SRRS context into the S notation. If an error is detected, then an error message is displayed to the user (translation response), the control flow translation ok is set to False, and processing ends. Otherwise, the control flow translation ok is set to True.

Name: Scan Specification, number 1.1.2
Type (process or control): process
Inputs: specification unit (from file)
Outputs: tokens, parsing response, parsing ok
Description: The purpose of this process is to lexically analyze the specification unit in the file name. The process generates a sequence of tokens. If an error occurs in the scanning, an error
message is displayed to the user (parsing response), the control flow parsing ok is set to False, and scanning ends. Otherwise, the control flow parsing ok is set to True. The Unix programming tool lex is used to scan the specification.

6.3.3 Data Dictionary

Name: Expanded requirements

Type (data or control): data

Description: The expanded requirement item is a parse tree of the expanded stimulus response requirements.

Name: Expanded requirements ok

Type (data or control): control

Description: The Expanded requirements ok item is used to indicate whether the expansion of the SRRS requirements is successful (value = True) or not (value = False).

Name: Parsing ok

Type (data or control): control

Description: The Parsing ok item is used to indicate the request to parse a specification unit is successful (value = True) or not (value = False).
Chapter 6 Tool Support for SRRS and DSPEC

Name: Parsing request

Type (data or control): control

Description: The Parsing request item is used to indicate the user has requested to parse a specification unit.

Name: Parsing response

Type (data or control): data

Description: The parsing response item is used to present a detected parsing error to the user.

Name: DSPEC check ok

Type (data or control): control

Description: The DSPEC check ok item is used to indicate whether the validation checks performed on a logical condition is successful (value = True) or not (value = False).

Name: DSPEC context

Type (data or control): data
Description: The DSPEC context item is a parse tree of data objects and data types that are in the DSPEC notation.

Name: DSPEC context ok

Type (data or control): control

Description: The DSPEC context ok item is used to indicate whether the creation of the DSPEC context is successful (value = True) or not (value = False).

Name: SRRS context

Type (data or control): data

Description: The SRRS context is a parse tree of the data objects, data types, stimuli, and responses that are in the SRRS notation.

Name: SRRS requirements

Type (data or control): data

Description: The SRRS requirements is a parse tree of the SRRS requirements
Name: SRRS validation ok

Type (data or control): control

Description: The SRRS validation ok item is used to indicate whether the validation checks performed on the SRRS context and requirements are successful (value = True) or not (value = False).

Name: translation ok

Type (data or control): control

Description: The translation ok item is used to indicate whether the translation of the specification is successful (value = True) or not (value = False).

Name: Translation request

Type (data or control): control

Description: The Translation request item is used to indicate the user has requested to translate a specification unit into S.

Name: translation response
Chapter 6 Tool Support for SRRS and DSPEC

Type (data or control): data

Description: The translation response item is used to present a detected translation error to the user.

Name: user request

Type (data or control): control

Description:= parse request | validation request | translation request

Name: validation ok

Type (data or control): control

Description: The validation ok item is used to indicate whether the validation of the specification is successful (value = True) or not (value = False).

Name: Validation request

Type (data or control): control

Description: The Validation request item is used to indicate the user has requested to perform the validation checks on a specification unit.
Chapter 6 Tool Support for SRRS and DSPEC

Name: validation response

Type (data or control): data

Description: The validation response item is used to present a detected validation error to the user.

6.4 Implementation Summary

The tool support is developed using the Unix programming tools lex and yacc [Lev92] and the C programming language [Ker88]. The total development is approximately 19 KSLOCs (refer to Table 6.1).

<table>
<thead>
<tr>
<th>Tool or Language</th>
<th>SRRS KSLOCs</th>
<th>DSPEC engine KSLOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>lex</td>
<td>0.51</td>
<td>0.25</td>
</tr>
<tr>
<td>yacc</td>
<td>5.2</td>
<td>0.90</td>
</tr>
<tr>
<td>C</td>
<td>8.78</td>
<td>3.10</td>
</tr>
</tbody>
</table>

Table 6.1 Implementation Effort Summary

6.5 Future Enhancements

The following future enhancements may be added into the requirements specifications for the SRRS notation and subsequently the tool support:

1. Generate an error if every local name in stimuli section is not used in requirements section and performance section
2. Generate an error if not every local name in responses section is not used in requirements section and performance section
3. Generate an error if every source/type_name pair is not used in requirements section
4. Generate an error if every destination/type_name pair is not used in requirements section
5. Generate an error if a declaration is not used (type, subtype, constant, VSP, system state component, function, predicate) in specification unit

6. Generate an error if a requirement is duplicated

7. Support an interactive mode in addition to the file based input/output

8. Generate reports on requirements represented in SRRS notation

9. Generate reports on data relationships described

10. Develop a GUI for the tool support

6.6 Conclusions

The tool support developed for the SRRS notation supports requests to parse, validate, and translate a specification unit into S. The architecture selected is combines a pipeline style with a layered style. The pipeline style is a standard solution for this compiler-like tool. The layered style allows the re-use of the DSPEC engine for other notation and tool development efforts.

The SRRS notation is shown to be a formal notation by the ability to translate the specifications written in SRRS into S. The S translations typecheck with version 2.5.1 of the FUSS tool.

The ability to translate the specification units written in the SRRS notation into S supports the automated generation of system level test specifications directly from the requirements [Don98]. This automation offers a reduction in the time to generate the test specifications and improves the quality of the test specifications generated. The process is automated which removes the time to manually generate the test specifications and averts the human errors introduced in the manual process.

In total, the development effort for SRRS and the DSPEC engine is approximately 19
KSLOCs of source code. The flexibility in the SRRS notation requires the development of a YACC source file that exceeds 5 KSLOCs lines.
Chapter 7  Experimental Evaluation of SRRS

7.1 Introduction

Every new technique needs to be objectively evaluated to determine its costs and benefits. There are several ways to evaluate a technique including case studies, pilot projects, or experiments. In this work, an experimental evaluation is selected to objectively evaluate the SRRS technique. An overview of the experimental design, results, and the conclusions based on the experimental data are presented in this chapter. The results are very encouraging and indicate the SRRS technique is a cost-effective way to develop requirement specifications. The time to write, review, and correct the specification is reduced as are the number of defects detected in a peer review process. There is, however, an increase in the training time for the authors.

This chapter overviews the experimental design used to evaluate the SRRS notation in Section 7.2 and presents a summary of the results in Section 7.3. Conclusions drawn from the experimental evaluation are described in Section 7.4. The complete experimental design and results are available in Appendix C.

7.2 Experimental Design

The problem statement for the experiment and an overview of the experimental design are presented in this section.

7.2.1 Problem Statement

The evaluation of requirements engineering techniques have been performed using case studies, pilot projects, and experiments [Bri94, Cra94, Lar96, Por95, Por98, San98]. Evaluations are used to discover strengths and weaknesses in a technique, determine the costs of applying a
Chapter 7 Experimental Evaluation of SRRS

technique, or determine the benefits of using a technique. The evaluations provide qualitative and/or quantitative results that can be used by project managers to decide if the technique is going to be used on a particular project [Bea91]. For new techniques developed in the academic world, such evaluations are important because the results may be used to encourage the transfer of the new technique to industry. The costs of introducing a formal notation include the cost of training employees in the tools and in the formal notation. The benefits include the availability of tools to assist the authors in automatically parsing, typechecking, and analyzing the specifications. With this tool support the author can detect and correct defects earlier in the develop lifecycle and reduce the cost of developing the software. These costs and benefits are subjective unless data is rigorously gathered and analyzed through empirical studies. These studies are expensive and time consuming to perform.

The purpose of this experiment is to objectively evaluate the costs and benefits of using a formal version of the SRRS notation along with its tool support in comparison to a similar, semi-formal version. The costs of introducing a formal notation include the cost of training employees in the tools and in the formal notation. To measure the costs, the amount of time spent in classroom and on the job training is recorded. The benefits include the availability of tools to assist the authors in automatically parsing, typechecking, and analyzing the specifications. The tool support is expected to reduce the time to develop the specifications and improve their quality. To measure the benefits, the amount of time used to write, review, and correct the specifications is recorded in addition to the number and type of defects recorded in the peer review process. In summary, the two techniques are compared in terms of the quality of the specifications written (number and category of defects detected) and the effort required to write the specifications (training, writing, reviewing, and correcting specification units). More rigorously, the objectives
of the evaluation are to test the following hypotheses:

1. Ho: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has the same average detected defect rate per allocated requirement object than a notation that does not use a completely defined syntax and automated tool support.

Ha: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has a lower average detected defect rate per allocated requirement object than a notation that does not use a completely defined syntax and automated tool support.

2. Ho: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has the same average effort per allocated requirement object to describe than a notation that does not use a completely defined syntax and automated tool support.

Ha: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has a lower average effort per allocated requirement object to describe than a notation that does not use a completely defined syntax and automated tool support.

3. Ho: The use of a notation with a completely defined syntax and automated tool support results in the same average training time per subject than a notation that does not use a completely defined syntax and automated tool support.
Ha: The use of a completely defined syntax and automated tool results in a higher average training time per subject than a notation that does not use a completely defined syntax and automated tool support.

The average detected defect rate is defined in terms of the total number of detected defects in the specification units divided by the total number of requirement objects allocated to the specification units written.

The average effort per allocated ROID is the total amount of time in minutes used to write, review, and correct the specification units written divided by the number of requirement objects allocated to the specification units written.

The average training effort per subject is the amount of time in minutes used to train the subjects in the notation both formally (in the classroom) and informally (independent study) divided by the number of subjects.

### 7.2.2 Experimental Design Overview

The overall structure of the experimental design is based on the lab package made available by [Por95] and has been tailored to meet the needs of this experiment. In this experiment, the treatment variable is the requirements specification technique used to write the specification units. The control group uses a semi-formal notation and the experimental group uses a formal notation with its tool support. The experiment evaluates the effort required (training time, writing time, reviewing time, correcting time) and the quality (total number and category of the defects found) in developing a set of specification units.

As in [Por95], the experiment begins with a formal training session for the subjects. As part
of the formal training, a hands-on practice exercise is given. When the formal training is complete, the subjects begin work on the experimental exercise.

The formal training for both the control and the experimental groups is structured as a set of four modules that are presented in a lecture format. Module one provides a brief introduction to the Formalware project (to provide the context for the work), reviews the waterfall and spiral software development lifecycle models, and describes the characteristics of a "good" requirement. Module two covers the initial phases of the software development lifecycle. It clarifies the distinction between a system level specification (SLS) and a software requirements specification (SRS), describes the process to develop a SRS, and reviews traceability. Module three introduces the requirements specification notation used by the group and describes the six step process used to describe the functional requirements in an SRS (refer to Figure 7.1). Module four reviews the requirements to system level testing interface and emphasizes the impact the quality of an SRS has on the development of system level test cases.

Modules one, two, and four are exactly the same for the control and the experimental groups; however, module three is tailored. The third module in the training material for the control group introduces the semi-formal stimulus response requirements specification notation to describe the functional requirements in an SRS. The training material for the experimental group introduces the formal stimulus response requirements specification notation. The six step process used to develop the specification units differs only in the use of the SRRS tool support by the experimental group in steps three and five.

The six step process is a top down refinement process for writing a specification unit. The inputs for the first step in the process are the SLS, the title of the specification unit to be
developed, and a list of ROIDs that are allocated to the title. The first step in the process is to ensure that each of the allocated ROIDs should be allocated and to determine if any ROIDs are missing from the allocation. After the ROIDs are checked, the author drafts an outline of the
specification unit in step two. The section headers are entered and each section is outlined in point form. In the third step, the author completes the version of the specification unit by refining the outline and filling in the details of the specification unit. In addition to completing the first version, a traceability report is generated to indicate how the allocated ROIDs have been addressed in the specification unit. The specification unit and the traceability report are submitted for a peer review in step four. The author’s peers review the specification unit and record defects according to type, category, defect checklist identifier, and a brief description. When the peer review is complete the specification unit, traceability report, and the defect recording sheet(s) are returned to the author. The author corrects the specification unit and revises the traceability report as necessary in step five. The review cycle is complete in the experiment when the specification unit has no defects detected or when the review cycle has occurred twice. The final version is submitted in the last step of the process.

The formal training session also includes a hands-on practice exercise. The exercise is based on a video rental system example in which patrons can rent and return video tapes. The subjects develop the Rent Tape specification unit using the notation and the six step process described in the third training module. The example and sample exercise are exactly the same for control and the experimental groups.

A package of support materials is provided at the beginning of the formal training session to each subject. The package consists of the four module training course, samples of the forms used to collect data, the checklist used to assist data collection, the SLS for the video rental system example, and the titles of the identified specification units and their allocated requirements for the video rental system exercise. The experimental group is also provided with a copy of the
Chapter 7 Experimental Evaluation of SRRS

SRRS user manual and the tutorial.

Two forms are used to assist in the collection of the experimental data: time and defect recording sheets. Time recording sheets are used to record the subject name, date, specification unit identifier, specification version number, and the amount of time spent on an activity. The possible activities are formal training, informal training, writing version 1, peer review, revising version, and "other". Defect recording data sheets are used to record the subject name, date, specification unit identifier, specification version number, and the defects detected. The defects are reported with an identification number, type of error, category of error, checklist identifier, and a textual description. A checklist of different defects is provided to assist the reviewer (Refer to Appendix C for samples of the data collection forms and the defect checklist).

The experimental exercise uses a SLS derived from a request for proposal for an integrated on-line library system [Cor87]. The SLS consists of 652 requirement objects. From this SLS, 52 specification units are identified and 432 software requirements are allocated to them. 18 of the specification units are randomly selected and written by both the control and the experimental groups.

7.3 Experimental Results

A summary of the experimental results is presented in Tables 7.1 - 7.3. The complete results are available in Appendix C.

7.3.1 Defect Rates

The hypotheses for this test are:
Ho: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has the *same average detected defect rate per allocated requirement object* than a notation that does not use a completely defined syntax and automated tool support.

Ha: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has a *lower average detected defect rate per allocated requirement object* than a notation that does not use a completely defined syntax and automated tool support.

The experimental results for the defect rates are summarized in Table 7.1. The results show a reduction in the syntax, type, and analysis defects for Group 2. The % difference between the two groups using the semi-formal notation (Group 1) and the formal notation (Group 2) for the total number of defects per allocated ROID shows an 81% reduction in detected defects.

<table>
<thead>
<tr>
<th></th>
<th>Number of syntax defects/ROID</th>
<th>Number of type defects/ROID</th>
<th>Number of analysis defects/ROID</th>
<th>Total number of defects/ROID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>0.99</td>
<td>0.74</td>
<td>0.88</td>
<td>2.61</td>
</tr>
<tr>
<td>Group 2</td>
<td>0.09</td>
<td>0.01</td>
<td>0.39</td>
<td>0.49</td>
</tr>
<tr>
<td>% Difference</td>
<td>-90.91</td>
<td>-98.65</td>
<td>-55.68</td>
<td>-81.23</td>
</tr>
</tbody>
</table>

Table 7.1 Summary of Defects Recorded

The test statistics used to evaluate the training time hypothesis are a 2 sample t-test and a p-value calculation. The calculations, performed with the tool Minitab, are summarized below.

**Paired T Test and C1: C2, C1**

**Paired T for C2-C1**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>SE Mean</th>
</tr>
</thead>
</table>

153
Chapter 7  Experimental Evaluation of SRRS

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>3</td>
<td>0.582</td>
<td>0.1823</td>
<td>0.106</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>2.373</td>
<td>0.797</td>
<td>0.536</td>
</tr>
<tr>
<td>Difference</td>
<td>3</td>
<td>-2.215</td>
<td>0.928</td>
<td>0.460</td>
</tr>
</tbody>
</table>

95% upper bound for mean distance: -0.871

T-test of mean =0 (vs >): T-value = -4.81 P-value = 0.020

A P-value < 0.05 indicates the results have statistical significance. The results support rejecting the null hypothesis.

7.3.2 Training Time

The hypotheses for this test are:

Ho: The use of a completely defined syntax and an automated parsing/typechecking/semantic analysis tool results in the same average training time per subject than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

Ha: The use of a completely defined syntax and an automated parsing/typechecking/analysis tool results in a higher average training time per subject than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

The training time is a metric of interest for individuals considering the use of the formal notation in comparison to its semi-formal notation. In this experiment, the formal training time includes the time the subjects are in the lecture style format plus the amount of time they spend on the hands-on exercise. In addition, the amount of time it takes the subjects in the experimental
group to work through the tutorial for the SRRS tool is considered as formal training. The formal training time is recorded as the number of minutes of formal training per author. The informal training includes the time the subjects use to review their training material or ask questions about the notation as they write the specification units. The informal training is recorded with respect to the number of allocated ROIDs. The total training time recorded is the sum of the formal training time and the informal training time per author. The experimental results for training time are summarized in Table 7.2. They show an increase in both the formal and informal training time for Group 2. The % difference between Group 1 and Group 2 is a 186% increase in total training time.

<table>
<thead>
<tr>
<th></th>
<th>Formal Training Time minutes/author</th>
<th>Informal Training Time minutes/ROID</th>
<th>Total Training Time minutes/author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>420.00</td>
<td>0.32</td>
<td>448.33</td>
</tr>
<tr>
<td>Group 2</td>
<td>835.00</td>
<td>5.02</td>
<td>1285.00</td>
</tr>
<tr>
<td>% Difference</td>
<td>98.81</td>
<td>1468.75</td>
<td>186.62</td>
</tr>
</tbody>
</table>

Table 7.2 Summary of Training Time

The test statistics used to evaluate the training time hypothesis are a 2 sample t-test and a p-value calculation. The calculations, performed with the tool Minitab, are summarized below:

2-Sample T for C1 vs C2

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>3</td>
<td>1245</td>
<td>368</td>
<td>213</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>448.3</td>
<td>10.4</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Difference = mu C1 - mu C2

Estimate for difference: 837
Chapter 7 Experimental Evaluation of SRRS

95% lower bound for distance: 383

T-test of mean = 0 (vs >): T-value = 3.93 P-value = 0.009 DF = 4

Both use pooled StdDev = 261

A P-value < 0.01 indicates the results have high statistical significance. The results support rejecting the null hypothesis.

7.3.3 Effort

The hypotheses for this test are:

Ho: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has the same average effort per allocated requirement object to describe than a notation that does not use a completely defined syntax and automated tool support.

Ha: The use of a notation with a completely defined syntax and automated tool support results in a requirement specification that has a lower average effort per allocated requirement object to describe than a notation that does not use a completely defined syntax and automated tool support.

The effort is a metric of interest for individuals considering the use of the formal notation in comparison to its semi-formal notation. The effort to write and put the specification units through a peer review is summarized in Table 7.3. The experimental results show a decrease in the amount of time to write and review requirements for Group 2. The % difference between Group 1 and Group 2 for the total amount of time to write and review requirements is a reduction
Chapter 7  Experimental Evaluation of SRRS

of 39%.

<table>
<thead>
<tr>
<th></th>
<th>Average time to write per ROID minutes</th>
<th>Average time to review and correct per ROID minutes</th>
<th>Average total time per ROID minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>17.58</td>
<td>15.28</td>
<td>32.86</td>
</tr>
<tr>
<td>Group 2</td>
<td>10.58</td>
<td>9.42</td>
<td>20.00</td>
</tr>
<tr>
<td>% Difference</td>
<td>-39.82</td>
<td>-38.35</td>
<td>-39.14</td>
</tr>
</tbody>
</table>

Table 7.3 Summary of Writing and Reviewing Time

The test statistics used to evaluate the training time hypothesis are a paired t-test and a p-value calculation.

Paired T Test and C1: C2 , C1

Paired T for C2-C1

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>3</td>
<td>21.4</td>
<td>7.4</td>
<td>4.3</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>36.4</td>
<td>18.6</td>
<td>10.7</td>
</tr>
<tr>
<td>Difference</td>
<td>3</td>
<td>-14.95</td>
<td>11.45</td>
<td>6.61</td>
</tr>
</tbody>
</table>

95% upper bound for mean distance: 4.35

T-test of mean = 0 (vs <): T-value = -2.26 P-value = 0.076

A P-value >0.05 indicates the results do not have statistical significance. The results do not support rejecting the null hypothesis.

7.4 Conclusions

The formal and informal training time both increased as expected for the Group using the formal notation in comparison to the group using the semi-formal notation. The additional burden of working through a tutorial, learning how the tool support works, and understanding the organi-
zation of the user manual all contribute to this increase.

The large increase in the informal training time is an interesting result. An explanation for this large increase is the additional complexity of having a concrete syntax and the tool support for the notation. Since the syntax must be conformed to and the validation checks all passed, the authors of the requirements may need to check the user manual, training material, and the tutorial documents more frequently as they work on the specifications.

The experimental results show a reduction in the number of defects detected in the peer review process. The concrete syntax in addition to the tool support that enforces the syntax and provides validation checks on the specification units allow the author to check their specifications before submitting them for review. Since only specifications that have a clean validation run (no errors are reported) with the tool support are allowed to go through the peer review process in the experiment, the reviewers receive a version that has had at least some of the defects removed. As a result the specification units have a lower detected defect rate and have a higher quality. The benefit of removing the machine detectable defects before the peer review is that the reviewers can concentrate on finding defects in the specification that need human analysis.

The experimental results also indicate there is a reduction in effort to write, review, and correct the specification units when using the formal notation in comparison to the semi-formal notation. The reduction can be attributed to the readability of the formal notation and the tool support. The reduced writing, review, and correction time indicates that the formal notation is at least as readable as the semi-formal notation. If the formal notation is not as readable, the time to write, review, and correct is expected to exceed the time when the semi-formal notation is used. The tool support allows the author to obtain feedback on syntax, type, and a small number of
analysis defects as the specification is being written. These defects can be removed before the specification is submitted for peer review. This reduces the number of defects in the specification making the remaining defects simpler and faster to identify in the peer review. A reduction in time is a benefit to the project, as it reduces the cost of developing the requirements specification.

The test statistics support rejecting the null hypotheses for the training time and the defects per ROID. The test statistics do not support rejecting the null hypothesis for the effort to develop a specification. One recommendation is to re-run the experiment with a larger sample size. The sample size for this run of the experiment is small and may be impacting the test statistics.

Application of Results

Given the experimental results, estimates can be made for the total time needed to train the authors and develop the specifications for a specific project. The caveat in the estimates made here is that the data used in the calculations is derived from one experimental project with 432 allocated ROIDs. The data has not been confirmed in projects of different sizes in different settings (i.e., academic vs. industrial setting).

Given the notation proposed, number of authors, and number of allocated ROIDs to be written the following calculation can be used to estimate the training time:

\[
\text{training time} = A \times T + R \times D \quad \text{(EQ 1)}
\]

where

A is the number of authors

T is the training time per author for a given notation
R is the number of ROIDs

D is the development time per ROID for a given notation.

For example, if the formal notation is used, the T is 835 minutes/author and D is 5.02 minutes/ROID.

Similarly, the experimental results can be used to estimate of the development time for a project. Given the notation proposed and number of allocated ROIDs to be written the following calculation can be used to estimate the development time:

\[
\text{development time} = R \times D
\]

where

R is the number of ROIDs

D is the development time per ROID for a given notation.

For example, if the formal notation is used, then D is 5.02 minutes/ROID.

The point at which it becomes feasible to use the formal notation can be estimated using the experimental results. For example, if a project has 2000 ROIDs and proposes to use 20 authors, the time to train, write, and review the specifications can be estimated for both the formal and semi-formal notations. The total time estimate is calculated as the sum of the training time (EQ1) and the development time (EQ2).

\[
\text{total time} = \text{training time} + \text{development time}
\]

Using the semi-formal notation, the calculation of the total time is:
training time = $20 \times 420 + 2000 \times 0.32$  
= 9040 minutes \hspace{1cm} (EQ 4)

development time = $2000 \times 32.86$  
= 65720 minutes \hspace{1cm} (EQ 5)

total time = 9040 + 65720  
= 74760 minutes \hspace{1cm} (EQ 6)

Using the formal notation, the calculation of the total time is:

training time = $20 \times 835 + 2000 \times 5.02$  
= 26740 minutes \hspace{1cm} (EQ 7)

development time = $2000 \times 20.00$  
= 40000 minutes \hspace{1cm} (EQ 8)

total time = 9040 + 65720  
= 66740 minutes \hspace{1cm} (EQ 9)

With these two calculations the formal notation shows a savings in time of approximately three weeks. If the project is scaled up, then a more dramatic time savings is calculated. For example, if the number of authors and the number of requirements are scaled up by an order of magnitude and the calculations shown above are repeated, the formal notation has a calculated time savings of approximately seven months. The selection of the formal notation in this case offers significant cost savings to the project. If the project is scaled down by an order of magnitude, then the formal notation has a calculated time savings of approximately 13 hours. Given a small project involving a couple of authors and 200 requirements, there is a very slight advantage in selecting the formal notation.

The results of this experiment have quantified the costs and benefits of using a formal notation with tool support in comparison with a similar, semi-formal version of the notation. The costs are increased classroom and on the job training time. The benefits include a reduced effort to
write and review the specification units and a reduced defect rate. The experimental results support the use of the formal notation in that the additional costs of training (in terms of time) are overcome by the gains achieved in the reduction of the amount of time to write and review the specification units. The challenge that follows this work is to verify the experimental results by replicating the experiment, perhaps as done by [Por98, San98] and then determine if the formal notation with its tool support are feasible for use in industry by running a small pilot project.
Chapter 8  Conclusions & Future Work

8.1 Contributions to Knowledge

There are four main contributions to knowledge in this work. The first contribution is the definition of a new requirements specification notation, SRRS. This notation has been designed to support the specification of large systems with complex data and logical conditions in the requirements (e.g., air traffic control) using a readable, yet mathematically defined, notation. SRRS is a synthesis of many ideas and research areas. Due to the complexity of the problem, a divide and conquer approach has been used to define SRRS. SRRS uses the DSPEC notation to manage the complexity of the logical conditions and data expressions.

The second main contribution to knowledge is the definition of a re-usable foundation notation, DSPEC, that has been designed to evaluate logical conditions and data expressions for the user on request. The advantage DSPEC provides over other notations that are used to describe logical conditions and data expressions is that DSPEC has rules built into the notation that can use the context to build paths, or nested function applications, for the user. In other notations, such as a higher order logic, the user must build the function applications themselves. This can be time consuming and error prone.

The third contribution of this research is to define a process to develop the SRRS notation. The process has five steps and describes a systematic refinement of an existing notation. The first step is an evaluation of the existing notation for strengths and weaknesses. In the second step, the results are used to update the notation and correct some of the identified problems. When this is done, the updated notation is evaluated and error checks are identified in the third step. The fourth step is the formalization of the notation (defining the syntax and semantics) and defining the error
The fourth main contribution is the experimental evaluation of SRRS. As few objective evaluations of requirements specification notations are available in the literature, the work done in this research can be expected to promote the use of experimental evaluation in requirements engineering. The experiment has been designed, run, and analyzed. The results have indicated that the SRRS notation significantly reduces the number of defects in a specification (81%), reduces the development time for the specification (39%), and has a minimal additional cost of additional training (2 days instead of one day for its semi-formal version).

8.1.1 SRRS

The SRRS notation has evolved from a long history requirements specification notations. Notations that have been drawn on to define SRRS are summarized in Figure 4.1. This figure is replicated for convenience below (refer to Figure 8.1) Components of the SRRS notation can be seen from early specification techniques that describe requirements using an input (stimulus), processing, and output (response) style [Dav77]. From an early structured analysis technique [Dem79] the data dictionary is used as a mechanism to describe the data in the system and the composition relationships between data using BNF. The idea of using a structured natural language is also introduced in [Dem79] and templates are introduced in [Hen80]. A later structured analysis technique [War85] promotes an outside-in, or external, partitioning of the requirements and generalizes the data relationships described as data schema using ERDs. Influences from the formal methods such as Z and HOL [Spi88,Gor93], include providing a mathematical foundation which supports automated typechecking as well as the translation into
SRRS [Coo00]
- external partitioning of requirements
- use of defined syntax, format
- requirements described as stimulus, processing, response
- generalized descriptions of data relationships
- machine readable, automated tool support

Thread-Capability [Pai93]
- external partitioning of requirements
- use of data dictionary (BNF)
- use of highly structured English
- requirements described as stimulus, processing, response

Structured Analysis [War85]
- external partitioning of requirements
- data schema ERDs

Structured Analysis [Ros77a, Ros77b]
Software Cost Reduction (SCR) Tables
Survey paper [Dav77]
- requirements described in terms of inputs, processing, outputs

Templates [Hen80]

ERDs [Chen76]

Structured Analysis [Dem79]
- use of data dictionary (BNF)
- use of structured natural language

Z [Spi88]
HOL88 [Gor93]

Use Case [Jac92, Boo99]
- external partitioning of requirements
- natural language
- formatting constraints

Figure 8.1 Evolution and Influences on the SRRS notation
other notations. Use cases [Jac92] use the external partitioning of requirements and move the requirements specification notations toward a natural language notation with some formatting constraints (normal course of events is described first followed by one or more alternate courses). The external partitioning, stimulus, processing, response style, writing constraints (formatting and highly structured natural language), and the use of a data dictionary are synthesized into the Threads-Capabilities technique [Pai93]. In turn, the SRRS notation draws on the Threads-Capabilities techniques, ERD concepts, tabular specifications, natural language processing and compiler construction techniques. SRRS relies on the DSPEC notation to manage the complexity of data relationships and logical conditions.

The important characteristics of SRRS are summarized in below in terms of characteristics of “good” requirements specification [Dav93, IEE93] and the practical application of the SRRS notation:

- **Understandable.** There are a number of characteristics in SRRS that contribute to the requirements specifications being understandable. The English-like syntax of SRRS makes the requirements specifications readable, because the notation is familiar. The highly structured format of the requirements specification makes the specification readable, because they follow a consistent format (there are no surprises to the reader).

- **Concise.** The ability to group stimuli and responses in SRRS can be used by the user to describe the transformation of multiple inputs into multiple outputs using a single requirement. SRRS also supports a list format that allows the user to describe multiple
requirements that are applicable when a particular stimulus message is received and a tabular format that is suitable for describing complex, logical conditions in a concise format.

- Precise and Unambiguous. The mathematical interpretation of the requirements specification makes the requirements precise and unambiguous. Logical conditions and data references in the requirements are evaluated to determine if they are unambiguous.

- Testable. The SRRS notation encourages a testable, blackbox style with its stimulus response format, defined syntax, and defined semantics. Test specifications have been automatically generated using the TCG tool [Don98] from an SRRS requirements specification.

- Training. The training for SRRS is available in a course that takes approximately two days, including practice exercises that provide hands-on experience. A user manual and tutorial are also available for SRRS and the tool support.

- Cost Effective. The experimental evaluation of SRRS found that in comparison to using a semi-formal version of the notation that did not have tool support, the time to train the authors in the classroom and on the job increased by 186%. The authors using the formal notation have the additional burden of learning the concrete syntax and working through the tutorial for the notation. The experiment also shows a reduced effort to write, review, and correct the specifications as well as an improvement in the quality of the specifications generated. The effort was decreased by 39% and the number of defects detected in peer review decreased by 81%. The tool support provides
feedback to the authors on the specifications before they are reviewed by their peers. This is an application of a desk check step in [Hum95]. Overall, the costs of the additional training are outweighed by the benefits of the reduced development time and improved quality of the specifications.

- Estimation Metrics. Estimates of the total effort required to train the authors and generate the specifications are developed based on the experimental results. These estimation calculations are new and could be useful in developing project schedules.

8.1.2 DSPEC

The DSPEC notation is a typed notation that supports the description of data entities (data types, data subtypes, and data objects), the relationships between these data entities (functions and predicates), a simplified means of referencing data, and requests to evaluate data expressions and logical conditions. Commonly used data types and objects are built into the notation (e.g., units of time, distance, temperature, and volume, bool, TRUE, FALSE). Domain specific data entities, functions, and predicates may be declared by the user.

The DSPEC notation evaluates these data expressions and logical conditions at two levels. After DSPEC builds complete paths (nested function applications) for the user, the requests are type checked with respect to the context. This provides the user with feedback regarding type mismatches and missing declarations. The user can quickly detect and correct these type errors in a specification. The second evaluation determines whether the data references are unambiguous or not. In a complex data model, data entities may be used in multiple places and referencing the correct one may not be straightforward. The evaluation checks defined in DSPEC can provide the user with immediate feedback and assist them in developing an unambiguous data model.
8.1.3 Experimental Evaluation of SRRS

The results of this experiment have objectively quantified the costs and benefits of using a formal notation with tool support in comparison with a similar, semi-formal version of the notation. The costs are increased classroom and on the job training time. The benefits include a reduced effort to write and review the specification units and a reduced defect rate. The experimental results support the use of the formal notation in that the additional costs of training (in terms of time) are overcome by the gains achieved in the reduction of the amount of time to write and review the specification units. The challenge that follows this work is to verify the experimental results by replicating the experiment, perhaps as done by [Por98, San98] and then determine if the formal notation with its tool support are feasible for use in industry by running a small pilot project.

8.2 Future Work

There are many interesting areas to pursue based on this work. The results obtained from the experimental evaluation of the SRRS notation in a lab setting are very promising. Replicating the experiment is necessary to confirm the results and extensions to the experiment could include an evaluation of the notation in an industrial setting with the subjects selected from a pool of professional software developers. This would provide an indication of the feasibility of the notation for use in industry. A small pilot project in industry may be suitable for this step in the evaluation process.

In the development and evaluation of SRRS, possible modifications and extensions have been noted. These include changes to both simplify and generalize the notation:
1. Simplify the signal coupling mechanism. Questions in the training session indicate that the signaling coupling mechanism is difficult for trainees to understand and apply. The internal signals couple with “commit”, “update”, or “send” responses could be removed. The gene purpose “signal” could be used instead making the notation simpler.

2. Remove the special case of the “return” response verb for acceptance and rejection processing. The verb “send” can be used here making the notation simpler.

3. Add the verbs “create” and “delete”. Currently the notation supports the specification of reading and modifying data but does not support the specification of creating or deleting data. Adding these two verbs generalizes the notation.

4. Import data declarations and relationships from files in other formats. For example, importing data declarations and relationships from a file in ASN.1 may be convenient.

5. Provide options for the style of punctuation used on labelled lists. The outline style in the Chicago manual of style [Chi93], for example, does not use punctuation on each item in a labelled list.

The scope of this work has been restricted to working within one specification unit. The following extensions would expand the scope of the research to include working across two or more specification units:

1. Match the external and exported functions and predicates across specification units. This would ensure that every external function or predicate declaration is available in the specification unit title by being declared as an exported function or predicate.

2. Synthesize and analyze a data dictionary from the declarations in the individual specification units.
Chapter 8 Conclusions & Future Work

DSPEC could be re-used with the definition of a test specification notation. This test specification notation may be useful for users who need to develop test specifications for testing third party software, for example, but do not believe writing a requirements specification would add value to their development programme (refer to Figure 8.2)

Figure 8.2 Re-use of DSPEC as a Foundation Notation
Bibliography


Cra94 D. Craigen, S. Gerhart, and T. Ralston, "Case study: Darlington nuclear generating


Bibliography


Jac92  I. Jacobson, M. Christerson, P. Jonsson, G. and Overgaard, Object-Oriented Soft-
Bibliography


175
Bibliography


Ray00 Raytheon Systems Canada Limited, Richmond Facility, web site www.ray.ca/rsclrf.html


War85 Ward, P. and Mellor, S. Structured Development for Real-Time Systems, Yourdon

Appendix A. Definition of the Stimulus Response Requirements Specification Notation

1. Introduction

The stimulus response requirements specification (SRRS) notation is defined in this Appendix. The definition is the requirements specification of the notation and is not intended to constrain the design or implementation of the tool support for the notation.

The structure of this Appendix is as follows. The basic components of the notation are described in Section 2. User commands and error messages are provided in Section 3. The SRRS notation is introduced in Section 4. This introduction to the notation uses an example from an integrated on-line library system (IOLS) (refer to Appendix D). The requirements evaluation rules are described in Section 5. The detailed syntax and evaluation rules are provided in Section 6. Sections 7 and 8 overview the data expressions and logical conditions supported in SRRS via the DSPEC notation.

2. Basic Components of the SRRS Notation

The lexical elements, reserved keywords, built-in components, and the underlying concepts of the notation are described in this section.

2.1 Lexical Elements

The lexical elements in SRRS include comments, labels, integers, punctuation, system_names, and phrases.

- A comment is a string of printable ASCII characters that are delimited by the percent sign character.
- A label element is composed of one or two ASCII characters followed by a closing parenthesis character.
- A number element is an integer.
- A punctuation element may be a comma, period, semi-colon, full-colon, opening or closing brace, opening or closing square bracket, opening or closing parenthesis, opening or closing angle bracket, double quote, or the backslash character.
- The system_name element is a string of one or more printable ASCII characters that does not include punctuation characters or whitespace characters (tab, space, newline).
- A phrase element is composed of one or more printable ASCII characters that does not contain opening and closing braces, opening and closing square brackets, opening and closing parentheses, opening and closing angle brackets, the percent sign character, or the backslash character. A phrase that is delineated by angle brackets, square brackets, backslashes, or double quotes may contain whitespace characters. One or more con-
secutive whitespace characters in a delineated phrase are scanned as a single occurrence of the space character. Whitespace at the beginning and the end of a phrase (delineated or not) are removed.

The number of characters in a phrase, system_name, or comment must not exceed 1024 or include the EOF.

2.2 Reserved Keywords and Characters

The reserved keywords and characters in SRRS are described in Table 1:

<table>
<thead>
<tr>
<th>Reserved Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>acceptance</td>
</tr>
<tr>
<td>all</td>
</tr>
<tr>
<td>an</td>
</tr>
<tr>
<td>As</td>
</tr>
<tr>
<td>at</td>
</tr>
<tr>
<td>Associations</td>
</tr>
<tr>
<td>by</td>
</tr>
<tr>
<td>cancel</td>
</tr>
<tr>
<td>canceled</td>
</tr>
<tr>
<td>commit</td>
</tr>
<tr>
<td>committed</td>
</tr>
<tr>
<td>Components</td>
</tr>
<tr>
<td>Constants</td>
</tr>
<tr>
<td>Declarations</td>
</tr>
<tr>
<td>exactly</td>
</tr>
<tr>
<td>event</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserved Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
</tr>
<tr>
<td>]</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

Table 1: Reserved Keywords and Characters in SRRS
2.3 Built-in Components

The built-in types, sources, destinations, and constants in SRRS are described in Table 2:

<table>
<thead>
<tr>
<th>Built-in Type Names</th>
<th>Built-in Source, Destination Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bool&gt;</td>
<td>Operator</td>
</tr>
<tr>
<td>&lt;celsius&gt;</td>
<td>“Remote Operator”</td>
</tr>
<tr>
<td>&lt;day&gt;</td>
<td>“internal send signal”</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>“internal commit signal”</td>
</tr>
<tr>
<td>&lt;date&gt;</td>
<td>“internal update signal”</td>
</tr>
<tr>
<td>&lt;metre&gt;</td>
<td>Printer</td>
</tr>
<tr>
<td>&lt;millisecond&gt;</td>
<td>“external commit”</td>
</tr>
<tr>
<td>&lt;kilometre&gt;</td>
<td>“external update”</td>
</tr>
<tr>
<td>&lt;response message&gt;</td>
<td>Plotter</td>
</tr>
<tr>
<td>&lt;second&gt;</td>
<td>“internal signal”</td>
</tr>
<tr>
<td>&lt;millimetre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;source&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;minute&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;centimetre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;destination&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;hour&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;decimetre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;year&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day of month&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;millilitre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;month&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day of week&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;decilitre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day of year&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;litre&gt;</td>
<td></td>
</tr>
<tr>
<td>TRUE :&lt;bool&gt;</td>
<td>Built-in Constants</td>
</tr>
<tr>
<td>FALSE :&lt;bool&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Built-in Components in SRRS

2.4 The SRRS context

The context of a specification unit includes:

- data types. A data type is the name for a set of objects. For example, “integer” is the name of the data type that contains all the integers
- data subtypes. A subtype is the name for a subset of a data type. For example, “positive integers” is the name of a subtype that contains only the positive integers
- constants. A constant is a data object that has a value that does not change, for example, “TRUE”, “FALSE”, or a physical constant such as “pi”
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

- variable system parameters (VSPs). A VSP is a data object with a value that may be tailored for a specific installation of the system. When a VSP is used in a specification unit, however, it has a fixed value. An example of a VSP is “Maximum number of library items allowed for charge”

- system state components. The system state components are domain specific data objects the author needs to use in order to describe the system. For example, names of system state components in a library system are “ON SHELF” or “The search response is sorted alphabetically”

- stimulus response components. A stimulus response component is a subtype for a source name, destination name, source message, or destination message. For example, the name “Administration Operator” may be introduced as a subtype of the source name. Although they are subtypes, the stimulus response components are placed in their own subsection to make the inputs and outputs of the specification unit clear

- static associations, functions, and predicates. A static association is a template version of a local function or a local predicate declaration. Functions and predicates are categorized as local, external, or exported

2.5 Naming Rules

The rules for naming the type names, subtype names, constants, VSPs, system state components, stimulus response components, functions, and predicates are described in this section. The restrictions on the names allow the user to reference an item in the specification unit unambiguously.

- Type names must be uniquely identified and may not be used as a subtype name or a data object name

- Subtype names, including the stimulus response components, must be uniquely identified by their name and may not be used as a type name or a data object name

- Constants, VSPs, and system state components must be uniquely identified by their name and type. The name of a data object may be re-used if the type name is different for each data object using the same name

- Statics Associations, Functions and Predicates must be uniquely identified by the body, parameter placement, and return type

3. SRRS User Commands and Error Messages

This section describes the user command options available and the error messages displayed by SRRS. The examples use the syntax of the tool support developed to demonstrate the commands.
3.1 User Commands

3.1.1 Introduction

There are three options that are supported by the SRRS. The first option allows the user to parse a requirements specification unit written in the SRRS notation. The second option allows the user to apply all validation checks that apply to the requirements specification unit. The third option allows the user to translate a requirements specification unit written in SRRS into the S notation.

3.1.2 Parse Requirements Specification

The user may determine if the requirements specification unit conforms to the SRRS syntax. If the requirements specification conforms to the grammar, the tool does not display any messages. If the requirements specification does not conform to the grammar, an error message is reported to the user along with the line number the error is detected on. In the example below (Refer to Figure 3.1) the user is requesting to parse the specification in the file search_catalogue.vl. Error messages are documented in Section 3.2.

```
cascade<23>SRRS search_catalogue.vl
cascade<24>
a. Successful Parsing Request

cascade<24>SRRS search_catalogue.vl
ERROR: parse error detected near '1)'
   line: 6
cascade<25>
b. Unsuccessful Parsing Request
```

Figure 3.1 a. Successful Parsing Request, b. Unsuccessful Parsing Request

3.1.1 Validate Requirements Specification

The user may determine if the requirements specification unit conforms to the SRRS grammar. In the example below (refer to Figure 3.2), the user is requesting to parse the requirements specification in the file search_catalogue.vl and determine if the requirements pass the validation checks.

If the requirements specification conforms to the grammar and passes the validation
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

cascade<23>SRRS -v search_catalogue.vl
cascade<24>

a. Successful Validation Request

cascade<24>SRRS -v search_catalogue.vl
ERROR: declaration missing
    The the search response is sorted alphabetically
    line: 34
cascade<25>

b. Unsuccessful Validation Request

Figure 3.2 a. Successful Validation Request, b. Unsuccessful Validation Request

checks, then no message is displayed (Figure 3.2a). If the requirements specification does not conform to the grammar or does not pass a validation check, an error message is reported to the user along with the line number the error is detected (Figure 3.2b). In this example, the user has accidentally typed in the word “the” twice when using a constant on line 34. As a result, SRRS cannot find a declaration that matches and reports the error. Error messages are documented in Section 3.2.

3.1.4 Translate Requirements Specification into S

The user may translate a requirements specification unit into S. In the example below (Figure 3.3), the user is requesting to translate the requirements specification in the file search_catalogue.vl into S.

If the requirements specification conforms to the grammar, passes the validation checks, and translates to S, then no message is displayed (Figure 3.3a). If successful, the translation to S is written to an ASCII file called translate.s in the current working directory. If the file translate.s already exists, the old file is overwritten. The translate.s file passes typechecking using the S typechecking tool FUSS [1], version 2.51.

If the requirements specification does not conform to the grammar, pass a validation check, or translate to S, an error message is reported to the user and the program exits (Figure 3.3b). If the validation checks are successful, the only way a translation to S can fail is due to insufficient memory. Error messages are documented in Section 3.2.

3.2 Error messages

A failure on parsing, performing validation checks, or translating the requirements to S is reported to the user with a brief textual message. A line number is also reported when the error is
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

cascade<23>SRRS -t search_catalogue.v1
cascade<24>

a. Successful Translation Request

cascade<24>SRRS -t search_catalogue.v1
ERROR: declaration missing
The the search response is sorted alphabetically
line: 34
cascade<25>

b. Unsuccessful Translation Request

Figure 3.3.a. Successful Translation Request, b. Unsuccessful Translation Request

due to a problem in the requirements specification file. Line numbers are not reported when the error is due to insufficient memory.

Expected parsing, validation, and translation errors are described in Table 3. The error messages output to the user all begin with the word ERROR and indicate a problem with the input requirements specification or the detection of a resource limitation (memory). In the table, the error messages are sorted alphabetically. Error messages that begin with the words INTERNAL ERROR are unexpected errors and indicate the tool has entered an unexpected state. These errors should be reported to the system maintainer.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration missing</td>
<td>A data type or object is used that has not been declared and is not built-in.</td>
</tr>
<tr>
<td>destination, message declaration missing in response</td>
<td>The destination, message used has not been declared in the response group.</td>
</tr>
<tr>
<td>destination name declaration missing</td>
<td>The destination name used is not declared.</td>
</tr>
<tr>
<td>destination name is an internal signal</td>
<td>The destination name used is an internal signal. Only external destination names</td>
</tr>
<tr>
<td>destination name not used in response group</td>
<td>The destination name is not used in the response group.</td>
</tr>
</tbody>
</table>

Table 3: Error Messages in SRRS
### Table 3: Error Messages in SRRS

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duplicate list component</td>
<td>A group stimulus or response name has a duplicate list component. Duplicated list components are not allowed.</td>
</tr>
<tr>
<td>duplicate object</td>
<td>The object has already been declared. Duplicate objects are not allowed.</td>
</tr>
<tr>
<td>duplicate static association</td>
<td>The static association has already been declared. Duplicate static associations are not allowed.</td>
</tr>
<tr>
<td>duplicate stimulus response component</td>
<td>The stimulus or response component has already been declared. Duplicate stimulus response components are not allowed.</td>
</tr>
<tr>
<td>duplicate subtype name</td>
<td>The subtype name has already been declared as a subtype name. Subtype names must be unique in the context.</td>
</tr>
<tr>
<td>duplicate type name</td>
<td>The type name has already been declared. Type names must be unique in the context.</td>
</tr>
<tr>
<td>end of Overview section not found</td>
<td>The end of the Overview section cannot be detected.</td>
</tr>
<tr>
<td>file name missing</td>
<td>The command line interface requires a file name as an argument.</td>
</tr>
<tr>
<td>file not found</td>
<td>The input file name does not exist in the current working directory.</td>
</tr>
<tr>
<td>function already declared in exported function section</td>
<td>The function has already been declared in the exported function section. Duplicate functions are not allowed.</td>
</tr>
<tr>
<td>function already declared in external function section</td>
<td>The function has already been declared in the external function section. Duplicate functions are not allowed.</td>
</tr>
<tr>
<td>function already declared in local function section</td>
<td>The function has already been declared in the local function section. Duplicate functions are not allowed.</td>
</tr>
<tr>
<td>function or predicate declaration missing</td>
<td>A function or predicate has been applied but not declared and is not built-in.</td>
</tr>
<tr>
<td>incorrect switch provided</td>
<td>The switch provided by the user is not supported by the tool. The two switches supported are -v (validate) and -t (translate to S).</td>
</tr>
<tr>
<td>input string too long, limit 1024</td>
<td>A comment, phrase, or system name has exceeded the maximum length.</td>
</tr>
</tbody>
</table>
### Error Message and Error Description Table

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insufficient memory</td>
<td>The program does not have enough memory to run. The heap has been exhausted.</td>
</tr>
<tr>
<td>labelled lists must have two or more elements</td>
<td>A labelled list has been used that only has one element in the list.</td>
</tr>
<tr>
<td>logical condition not of type bool</td>
<td>A logical condition does not evaluate to type bool.</td>
</tr>
<tr>
<td>missing closing brace</td>
<td>A logical condition is missing a closing end brace.</td>
</tr>
<tr>
<td>multiple matches found</td>
<td>Multiple matches to a name have been found. The name is ambiguous.</td>
</tr>
<tr>
<td>multiple relationships found</td>
<td>Two or more ways of referencing the data have been found. The data reference is ambiguous.</td>
</tr>
<tr>
<td>no relationship found</td>
<td>Zero ways of referencing the data have been found by building paths. The data reference is ambiguous.</td>
</tr>
<tr>
<td>object already declared in constant section</td>
<td>The object has already been declared in the constant section. Duplicate object declarations are not allowed.</td>
</tr>
<tr>
<td>object already declared in type name section</td>
<td>The object name has already been declared as a type name. Type names must be unique in the context.</td>
</tr>
<tr>
<td>object already declared in subtype section</td>
<td>The object name has already been declared as a subtype name. Subtype names must be unique in the context.</td>
</tr>
<tr>
<td>object already declared in vsp section</td>
<td>The object has already been declared in the vsp section. Duplicate object declarations are not allowed.</td>
</tr>
<tr>
<td>object already declared in system state component section</td>
<td>The object has already been declared in the system state component section. Duplicate object declarations are not allowed.</td>
</tr>
<tr>
<td>parameter must be angle bracket delimited</td>
<td>A parameter in a function or predicate declaration is not delimited by angle brackets.</td>
</tr>
<tr>
<td>parse error detected near</td>
<td>A general error message that indicates the requirements specification does not conform to the grammar.</td>
</tr>
<tr>
<td>predicate already declared in exported predicate section</td>
<td>The predicate has already been declared in the exported predicate section. Duplicate predicates are not allowed.</td>
</tr>
<tr>
<td>predicate already declared in external predicate section</td>
<td>The predicate has already been declared in the external predicate section. Duplicate predicates are not allowed.</td>
</tr>
</tbody>
</table>

Table 3: Error Messages in SRRS
### Appendix A. Definition of the Stimulus Response Requirements Specification Notation

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate already declared in local predicate section</td>
<td>The predicate has already been declared in the local predicate section. Duplicate predicates are not allowed.</td>
</tr>
<tr>
<td>request message declaration missing</td>
<td>The request message used has not been declared.</td>
</tr>
<tr>
<td>response group declaration missing</td>
<td>The response group name used has not been declared.</td>
</tr>
<tr>
<td>response message declaration missing</td>
<td>The response message used has not been declared.</td>
</tr>
<tr>
<td>response group used does not match the declaration in the responses section</td>
<td>The response group used does not match in the response action to its declaration.</td>
</tr>
<tr>
<td>return type of a function cannot be bool</td>
<td>The function declaration cannot have return type bool.</td>
</tr>
<tr>
<td>source message declaration missing</td>
<td>The source message used has not been declared.</td>
</tr>
<tr>
<td>source, message declaration missing in stimulus group</td>
<td>The source, message used has not been declared in the stimulus group.</td>
</tr>
<tr>
<td>source name declaration missing</td>
<td>The source name used has not been declared.</td>
</tr>
<tr>
<td>source name not used in stimulus group</td>
<td>The source name is not used in the stimulus group.</td>
</tr>
<tr>
<td>static association must related types</td>
<td>The static association declared has a data object name. The association must be between data types.</td>
</tr>
<tr>
<td>static association not relating two unique types</td>
<td>The static association declared has the type name associated with itself. The two types must be different.</td>
</tr>
<tr>
<td>stimulus group declaration missing</td>
<td>The stimulus group name used is not declared.</td>
</tr>
<tr>
<td>stimulus response component already declared in constant section</td>
<td>The stimulus or response component has already been declared in the constant section.</td>
</tr>
<tr>
<td>stimulus response component already declared in subtype section</td>
<td>The stimulus or response component has already been declared in the subtype section.</td>
</tr>
<tr>
<td>stimulus response component already declared in system state component section</td>
<td>The stimulus or response component has already been declared in the system state component section.</td>
</tr>
<tr>
<td>stimulus response component already declared in type section</td>
<td>The stimulus or response component has already been declared in the type section.</td>
</tr>
<tr>
<td>stimulus response component already declared in vsp section</td>
<td>The stimulus or response component has already been declared in the vsp section.</td>
</tr>
</tbody>
</table>

Table 3: Error Messages in SRRS
### Appendix A. Definition of the Stimulus Response Requirements Specification Notation

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype name already declared as a type name</td>
<td>The subtype name has already been declared as a type name. Subtype names must be unique in the context.</td>
</tr>
<tr>
<td>type mismatch</td>
<td>The actual type does not match the expected type.</td>
</tr>
<tr>
<td>unexpected group response name</td>
<td>The actual group response name does not match the expected group response name in a response qualification.</td>
</tr>
<tr>
<td>unexpected placement of parameter</td>
<td>A parameter is found in a response qualification but is not expected.</td>
</tr>
</tbody>
</table>

Table 3: Error Messages in SRRS
4. Introduction to a Requirements Specification Unit in SRRS

A specification unit written in SRRS introduces a unique section of a software requirements specification document. A specification unit is composed of seven sections: title, overview, stimuli, responses, requirements, declarations, and a performance section (Figure 4.1).

The purpose of each of these sections is summarized below.

4.1 Title Section

The title provides a unique identifier for the specification unit in the requirements specification document. The title should be a meaningful name that indicates the contents of the specification unit. The title is not a testable requirement.

4.2 Overview Section

The overview section provides a summary description of the rejection and normal processing performed by the specification unit. There are no testable requirements in the
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

Overview section.

4.3 Stimuli Section

The stimuli section documents all of the stimuli that trigger the specification unit. The stimuli section introduces unique group stimulus names. Each group stimulus name is the name for a list of one or more (source name, message) pairs. The source name indicates the origin of the stimulus message. The message indicates the contents of the stimulus. The group stimulus name provides a name for referencing one or more of the elements of the list in the requirements section in a compact notation.

4.4 Responses Section

The responses section documents all of the responses that are generated by the specification unit. The responses section introduces unique group response names. A group response name is the name for a list of one or more unique (destination name, message) pairs or (destination name, message, message) triples. The destination name indicates where the response is going. The message indicates the contents of the response. The group response name provides a name for referencing one or more of the elements of its list in the requirements section in a compact notation. The order of the responses is: rejection declaration, acceptance declaration, and lastly send, commit, update, and signal declarations.

4.5 Requirements Section

The requirement section describes the processing required to transform a stimulus into one or more responses. The requirements section is composed of one or more requirements in the following order: rejection requirements, acceptance requirements, followed by stimulus response requirements. A rejection requirement describes the generation of a rejection with reason response. For example, it may be used when a check has failed and an appropriate rejection message must be returned to the user indicating why the check failed. An acceptance requirement describes the generation of an acceptance response. The purpose of the acceptance requirement is to provide feedback to the user. A stimulus response requirement expresses either a relationship between a stimulus and one or more responses, or the generation of one or more responses under specified conditions (the stimulus is optional).

A stimulus response requirement can be extended by describing response qualification and/or response response requirements. A response qualification describes a postcondition on the requirement. A response response requirement is a compact notation for additional stimulus response requirements. The evaluation of the stimulus response, response qualification, and response response requirements is presented in Section 5.

4.6 Declaration Section

The declaration section describes the context for the specification unit. The section introduces data types, data subtypes, static associations, constants, variable system parameters, system state components, stimulus response components, functions, and predicates. The functions and predicates are categorized as local (declared in this specification unit), external
(declared in another specification unit), or exported (declared in this specification unit for use in other specification units).

4.7 Performance Section

The performance section describes the performance requirements for the specification unit. The performance requirements are described as a list of zero or more logical conditions.
5. Requirement Evaluation Rules

The compact notation supported by the SRRS notation requires an evaluation (preprocessing) in order to represent all the requirements explicitly. The stimulus response matching rules, response response requirement, and response qualification requirement evaluation rules are described in the following sections. The explicit representation of all of the requirements is necessary for validation checking and the translation to S.

5.1 Stimulus Response Matching Rules in SRRS

The requirement statements relate a group of stimuli that trigger the processing to one or more groups of responses using a compact notation. The stimuli and responses must be matched, however, to represent all of the requirements. Three sets of rules for generating the stimulus response pairs are described in this section. The kind of response determines which set of rules are used to evaluate the stimulus response requirements.

5.1.1 Return Response

There are four cases to consider when a stimulus is matched with either a rejection with reason or an acceptance return response. The cases are based on whether a source and/or the destination name are qualified, or specified, in the requirement. The first case has no qualification on the source or destination. The second case has a qualified source and a unqualified destination. The third case has an unqualified source and a qualified destination. The fourth case has a qualified source and qualified destination. The cases are illustrated below using the acceptance return response. The matching rules are the same for the rejection with reason return response. Each case begins with an illustrative example and is followed by the pseudocode to match, or pair, the stimulus and response into a requirement and a brief summary of the rules.

Case One
Example:
Upon receipt of a [group stimulus name], the system shall return an [acceptance].

Pseudocode:
For each stimulus in the [group stimulus name]
    Obtain the source name
    For each response in the [acceptance]
        If the source name is not an internal signal
            If the destination name matches the source name,
                pair the stimulus and response
            Else
                pair the stimulus and response
        Else
            pair the stimulus and response

In case one, for externally visible stimuli, a stimulus is paired with every response that has the same destination name as the source name. For internally signalled stimuli, a stimulus is paired with every response.

Case Two
Example:
Upon receipt of a [group stimulus name] from X, the system shall return an [acceptance].

Pseudocode:
For each stimulus in the [group stimulus name]
   Obtain the source name
   If the source name is X then
      For each response in the [acceptance]
         If the source name is not an internal signal
            If the destination name matches the source name,
               pair the stimulus and response
         Else
            pair the stimulus and response

   In case two, for externally visible stimuli, a stimulus with source name X is paired with every response with destination name X. For internally signalled stimuli, a stimulus with source name X is paired with every response.

Case Three

Example:
Upon receipt of a [group stimulus name], the system return an [acceptance] to Y.

Pseudocode:
For each stimulus in the [group stimulus name]
   Obtain the source name
   If the source name is Y then
      For each response in the [acceptance]
         If the destination name is Y
            pair the stimulus and response

   In case three, a response with destination name Y is paired with every stimulus with source name Y.

Case Four

Example:
Upon receipt of a [group stimulus name] from X, the system shall return an [acceptance] to Y.

Pseudocode:
For each stimulus in the [group stimulus name]
   Obtain the source name
   If the source name is X then
      For each response in the [acceptance]
         If the destination name is Y
            pair the stimulus and response
In case four, a response with destination name Y is paired with every stimulus with source name X.

## 5.1.2 Send Response

There are six cases to consider when a stimulus is matched with a send response. Cases one through four are for requirements that have a stimulus present. Cases five and six are for requirements that do not have a stimulus present (the stimulus is optional). The first case has no qualification on the source or destination. The second case has a qualified source and an unqualified destination. The third case has an unqualified source and a qualified destination. The fourth case has a qualified source and qualified destination. The fifth case has an unqualified destination. The sixth case has a qualified destination. Each case begins with an illustrative example and is followed by the pseudocode to match, or pair, the stimulus and response into a requirement and a brief summary of the rules.

**Case One**

**Example:**
Upon receipt of a [group stimulus name], the system shall send a [group response name].

**Pseudocode:**
For each stimulus in the [group stimulus name]
    Obtain the source name
    For each response in the [group response name]
        If the source name is not an internal signal
            If the destination name matches the source name,
                pair the stimulus and response
            pair the stimulus with the internal signals of the response
        Else
            pair the stimulus and response
            pair the stimulus with the internal signals of the response

In case one, for externally visible stimuli, a stimulus is paired with every response that has the same destination name as the source name. For internally signalled stimuli (internal send signal, internal update signal, internal commit signal, internal signal), a stimulus is paired with every response.

**Case Two**

**Example:**
Upon receipt of a [group stimulus name] from X, the system shall send a [group response name].

**Pseudocode:**
For each stimulus in the [group stimulus name]
    Obtain the source name
    If the source name is X then
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

For each response in the [group response name]
  If the source name is not an internal signal
    If the destination name matches the source name,
      pair the stimulus and response
      pair the stimulus with the internal signals of the response
  Else
    pair the stimulus and response
    pair the stimulus with the internal signals of the response

In case two, for externally visible stimuli, a stimulus with source name X is paired with every response with destination X and the associated internal responses. For internally signalled stimuli, a stimulus with source name X is paired with every response.

Case Three

Example:
Upon receipt of a [group stimulus name], the system shall send a [group response name] to Y.

Pseudocode:
For each stimulus in the [group stimulus name]
  Obtain the source name
  If the source name is Y then
    For each response in the [group response name]
      If the destination name matches the source name
        pair the stimulus and response
        pair the stimulus with the internal signals of the response

In case three, a response with destination name Y and the associated internal responses are paired with every stimulus with source name Y.

Case Four

Example:
Upon receipt of a [group stimulus name] from X, the system shall send a [group response name] to Y.

Pseudocode:
For each stimulus in the [group stimulus name]
  Obtain the source name
  If the source name is X then
    For each response in the [group response name]
      If the destination name is Y
        pair the stimulus and response
        pair the stimulus with the internal signals of the response

In case four, a response with destination name Y and the associated internal responses are
paired with every stimulus with source name X.

Case Five

Example:
If (logical condition), the system shall send a [group response name].

Pseudocode:
For each response in the [group response name]
generate a stimulus response requirement
(stimulus is null)

In case five, a stimulus response requirement is generated for each response in the group response name.

Case Six

Example:
If (logical condition), the system shall send a [group response name] to Y.

Pseudocode:
For each response in the [group response name]
If the destination name is Y
  generate a stimulus response requirement
  (stimulus is null)
  generate a stimulus response requirement for the
  internal signals of the response
  (stimulus is null)

In case six, a stimulus response requirement is generated for each response with destination name Y and the associated internal responses.

5.1.3 Commit, Update, or Signal Response

There are three cases to consider when a stimulus is matched with a commit signal, update signal, or signal response. The first case has no qualification on the source or destination. The second case has a qualified source and an unqualified destination. The third case has no stimulus and an unqualified destination. The matching rules are the same for commit, update, and signal responses. The commit response is used in the cases below. Each case begins with an illustrative example and is followed by the pseudocode to match, or pair, the stimulus and response into a requirement and a brief summary of the rules.

Case 1

Example:
Upon receipt of a [group stimulus name], the system shall commit a [group response name].

Pseudocode:
For each stimulus in the [group stimulus name]
  For each response in the [group response name]
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

pair the stimulus and response
pair the stimulus with the internal signals of the response

In case one, every stimulus is paired with every response.

Case 2

Example:
Upon receipt of a [group stimulus name] from X, the system shall commit a [group response name].

Pseudocode:
For each stimulus in the [group stimulus name]
  If the source name is X
    For each response in the [group response name]
      pair the stimulus and response
      pair the stimulus with the internal signals of the response

In case two, every stimulus with source name X is paired with every response.

Case 3

Example:
If {logical condition}, the system shall commit a [group response name].

Pseudocode:
For each response in the [group response name]
  generate a stimulus response requirement
    (stimulus is null)
  generate a stimulus response requirement for the internal signals of the response
    (stimulus is null)

In case three, the (null) stimulus is paired with every response.

5.2 Response Qualification Evaluation Rules

A stimulus response requirement followed by one or more response qualification requirements is a stimulus response requirement with one or more postconditions associated with the response. If there are two or more response qualification requirements present, they are evaluated as a conjunction. The response qualifications may be a data object of type bool, a subtype of bool, the type bool, or a predicate application.

5.3 Response Response Evaluation Rules

A stimulus response requirement followed by one or more response response requirements is a compact notation for two or more stimulus response requirements. When a response response requirement is evaluated, the compact notation may be expanded in three ways. The first expansion is that the stimulus on the stimulus response requirement, if present, is
added on to each of the response response requirements. The second expansion is that the logical condition on the stimulus response requirement, if present, is added on to each of the response response requirements. The third expansion is that the logical condition on each of the response requirements, if present, is added to each of its subsequent response response requirements.

5.4 Requirement Evaluation Example

The example illustrates the application of the stimulus response matching rules for a send response and the response qualification evaluation rules. The stimulus, response, requirement, and a system state component declaration used in the evaluation of the example are presented in the SRRS notation. Subsequently, the actual requirements represented in the example are presented.

Stimuli:

1) The IOLS shall satisfy the requirements described below upon receipt of a [search request] from:
   a) Operator <search request>;
   b) “Remote Operator” <search request>;
   c) VPL <search request>;
   d) SFU <search request>;
   e) UVIC <search request>;
   f) EPL <search request>.

Responses:

2) As specified by the requirements described below, the IOLS shall send a [search response] to:
   a) Operator <search response>;
   b) “Remote Operator” <search response>;
   c) VPL <search response>;
   d) SFU <search response>;
   e) UVIC <search response>;
   f) EPL <search response>.

Requirements:

2) Upon receipt of a [search request], the IOLS shall send a [search response]. {The search response is sorted alphabetically}. 198
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

System State Components:
1) “The search response is sorted alphabetically”: <bool>.

In the requirement written above, the stimulus response requirement is “Upon receipt of a [search request], the IOLS shall send a [search response]”. The evaluation of this requirement is based on the send response matching rules for the case that has an unqualified source and destination.

Following the stimulus response requirement is the response qualification requirement “{The search response is sorted alphabetically}”. This response qualification is a data object of type bool that is declared in the system state component declaration subsection.

After evaluation, the six stimulus response requirements generated are:
1) Upon receipt of a Operator <search request>, the IOLS shall send a Operator <search response>. {The search response is sorted alphabetically}.
2) Upon receipt of a “Remote Operator” <search request>, the IOLS shall send a “Remote Operator” <search response>. {The search response is sorted alphabetically}.
3) Upon receipt of a VPL <search request>, the IOLS shall send a VPL <search response>. {The search response is sorted alphabetically}.
4) Upon receipt of a SFU <search request>, the IOLS shall send a SFU <search response>. {The search response is sorted alphabetically}.
5) Upon receipt of a UVIC <search request>, the IOLS shall send a UVIC <search response>. {The search response is sorted alphabetically}.
6) Upon receipt of a EPL <search request>, the IOLS shall send a EPL <search response>. {The search response is sorted alphabetically}.

6. Syntax and Validation Checks for a Requirements Specification in SRRS

A requirements specification unit introduces a unique section of a software requirements specification document. A specification unit describes the title, overview, stimuli, responses, requirements, declarations, and performance for one section. These seven sections for a requirements specification unit are described in detail below. For each section, a synopsis, concrete syntax, and the validation checks that may be performed are provided.

6.1 General Remarks

The following sections of a specification unit are global with respect to the software requirement specification document: Title Section, Exported Functions Subsection, and Exported Predicates Subsection. All other sections are local to the specification unit. The built-in types and
objects (e.g., <bool>, TRUE) are available for use in every specification unit.

The BNF conventions used in the definition of the language are listed below:
1. Reserved symbols are shown in bold-face type, e.g., the.
2. Non-terminals are underlined, e.g., stimulus response requirement.
3. Terminals, or tokens, are in plain text, e.g., Operator.
4. A syntactic form enclosed within "[" and "]" indicates that the syntactic form is optional.
5. A syntactic form enclosed within "[" and "]" followed by a "*" indicates zero or more iterations of the syntactic form.
6. Required punctuation characters are enclosed within single quotes.
7. A syntactic form enclosed within "[" and "]" followed by a "+" indicates one or more iterations of the syntactic form.
8. The symbols "[", "]" used to specify syntactic forms should not be confused with the reserved symbols "{", "}". The latter are shown in bold-face in this document.

The subsections that defined the language are not applicable at all times. The definition should be read with the following rules:
1. If a specification passes the lexical analysis (i.e., it parses correctly), then the evaluation rules are applicable.
2. If the specification passes the application of the evaluation rules, then the specification may be translated into S.

Labelled lists are supported to a depth of two levels. The indexing labels used for the lists are defined by the project standard. The concrete syntax defines a label as one or two printable ASCII characters followed by a closing parenthesis. The indexing used in the hierarchy below is only a suggestion for use.
1. Label level one is a one digit number that identifies the list item. The first list item has the label "1)". Each n+1 list item has the number (n+1) followed by a closing parenthesis.
2. Label level two is a lower case letter that identifies the list item. The first list item has the label "a)". Each n+1 list item has the subsequent letter followed by a closing parenthesis. If more than 26 items are needed in a list, the 27th item begins with the label "aa)".

The punctuation used on the items in the labelled list is semi-colons at the end of each line and a period or semi-colon on the last line item.

In a requirement, a labelled list is a compact notation for writing multiple, similar requirements. In the example below, a requirement written in the list format and its equivalent requirements written in a non-list format are shown. The list format removes redundancy in the requirement which makes it simpler to modify.

Example using labelled list
1) Upon receipt of a [do something request], if {some condition holds } the some_system shall:
   a) send a [send response];
   b) update a [update response];
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

c) signal a [signal response].

Example not using labelled list
1) Upon receipt of a [do something request], if {some condition holds}, the
   some_system shall send a [send response].

2) Upon receipt of a [do something request], if {some condition holds}, the
   some_system shall update a [update response].

3) Upon receipt of a [do something request], if {some condition holds}, the
   some_system shall signal a [signal response].

6.2 Requirement Specification Unit

6.2.1 Synopsis

A requirements specification unit introduces a unique section of a software requirements
specification document. A specification unit describes the title, overview, stimuli, responses,
requirements, declarations, and performance for one section.

6.2.2 Concrete Syntax

A specification unit must match the following BNF rules:

```
specification unit :=
   Title Section
   Overview Section
   Stimuli Section
   Responses Section
   Requirements Section
   Declarations Section
   Performance Section
```

6.2.3 Validation Checking

The specification unit fails validation checking if any of its sections fail.

6.3 Title Section

6.3.1 Synopsis

The Title Section uniquely identifies the specification unit in the software requirements
specification. There are no testable requirements in the title section.
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

6.3.2 Concrete Syntax

A title section must match the following BNF rules:

Title Section :=
  Title : 'V phrase 'V

Example
Title: \My title\ 

6.3.3 Validation Checking

There are no validation checks performed on the title section.

6.4 Overview Section

6.4.1 Synopsis

The overview section describes the rejection and normal processing performed by the specification unit. There are no testable requirements in the title section.

6.4.2 Concrete Syntax

An overview section must match the following BNF rules:

Overview Section :=
  Overview : phrase

Example
Overview:
The My title specification unit does this, that, and the other thing.

6.4.3 Validation Checking

There are no validation checks performed on the overview section.

6.5 Stimuli Section

6.5.1 Synopsis

The stimuli section documents all of the stimuli that trigger the specification unit. The stimuli section is organized as a list of zero or more, unique group stimulus names. Each group stimulus name is the name for a list of (source, message) pairs. The source name indicates the origin of the stimulus message. The message name indicates the data type of the stimulus
message. The group stimulus name provides a name for referencing one or more of the elements of the list in the requirements section.

6.5.2 Concrete Syntax

A Stimuli Section must match the following BNF rules:

Stimuli Section :=
    Stimuli ':'
    { label The system_name shall satisfy the requirements described below upon receipt of a [' group_stimulus_name ']' from: labeled list of stimuli } *

labeled list of stimuli :=
    {
        ([ label source request message '; ' ])
        | ([ label internal commit signal of a signal message '; ' ])
        | ([ label internal update signal of a signal message '; ' ])
        | ([ label internal send signal of a signal message '; ' ])
        | ([ label internal signal of a signal message '; ' ])
    }*
    ([ label source request message '.' ]
    | ([ label internal commit signal of a signal message ''].
    | ([ label internal update signal of a signal message ''].
    | ([ label internal send signal of a signal message ''].
    | ([ label internal signal of a signal message ''].

group_stimulus_name := phrase
source := phrase | "" phrase "" | '<' phrase '>
request message := '<' phrase '>
signal message := '<' phrase '>

Example

Stimuli:
1) The some_system shall satisfy the requirements described below upon receipt of a [do this request] from:
   a) Operator <do this>;
   b) "Remote Operator" <do this too>;
   c) internal signal of a <do this as well>.

1) The some_system shall satisfy the requirements described below upon receipt of a [do that request] from:
   a) internal commit signal <do that>;
   c) internal signal of a <do that too>.

1) The some_system shall satisfy the requirements described below upon receipt of a [do the other thing request] from:
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

a) Operator <do that>.

6.5.3 Validation Checking

The stimuli section fails validation checking if:
1. a source is used that is not declared
2. a source is not of type "<source>"
3. a request_message is used that is not declared
4. a request_message is not of type "<request message>"
5. a signal_message is used that is not declared
6. the group_stimulus_name is not unique (i.e., the group_stimulus_name is already declared)
7. a source, request_message pair in a group_stimulus_name is not unique (i.e., there are duplicate components in the list of a group_stimulus_name)
8. an internal commit, update, send, or signal signal_message pair in a group_stimulus_name is not unique (i.e., there are duplicate components in the list of a group_stimulus_name)

6.6 Responses Section

6.6.1 Synopsis

The responses section documents all of the responses generated by the specification unit. The responses section is organized as a list of one or more unique group response names. The group response name is the name for a list of one or more unique (destination, message) pairs or (destination, message, message) triples. The destination indicates where the response is going. The message indicates the contents of the response. The group response name provides a name for referencing one or more of the elements of the list in the requirements section.

6.6.2 Concrete Syntax

A Responses Section must match the following BNF rules:

Responses Section :=
Responses:
( rejection with reason response { general response }*)
| ( acceptance response { general response }*)
| ( rejection with reason response acceptance response { general response }*)
| { general response }+

general response:
( send response | commit response | update response | signal response )

Example

Responses:
1) As specified by the requirements described below, the some_system shall return a [rejection with reason] to:
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

a) Operator <this rejection message>;
b) “Remote Operator” <that rejection message>.

2) As specified by the requirements described below, the some_system shall return an [acceptance] to:
   a) Operator <this acceptance message>;
   b) “Remote Operator” <that acceptance message>.

3) As specified by the requirements described below, the some_system shall send a [did something response] to:
   a) Operator <this to send>;
   b) internal send signal of a <this send signal>;
   c) “Remote Operator” <that to send>;
   d) Printer <the other thing to print>.

4) As specified by the requirements described below, the some_system shall commit a [did that commit response] as:
   a) <that data to commit>;
   b) internal commit signal of a <committed data signal>;
   c) internal commit signal of a <committed data signal as well>;
   d) <part of the data> to <a larger part of the data>.

5) As specified by the requirements described below, the some_system shall update a [did the other update response] as:
   a) <the other data to update>;
   b) <part of the data to update> to <a bigger part of the data>;
   c) <even more data to update>;
   d) internal update signal <even more data updated signal>.

6) As specified by the requirements described below, the some_system shall signal a [signaling some response] to:
   a) internal signal of a <something happened signal>.

6.6.3 Validation Checking

The response section fails validation checking if any of its component responses fail validation checking.

6.6.4 Rejection with Reason Response

6.6.4.1 Synopsis

The rejection with reason response introduces a unique group_response_name declaration “[ rejection with reason]”. The group_response_name is the name for a list of one or more destination, message pairs. The purpose of the rejection with reason response is to provide
feedback to the user when a check in the requirements has failed.

6.6.4.2 Concrete Syntax

A rejection with reason response must match the following BNF rules:

```
rejection with reason response :=
    label As specified by the requirements described below, the system_name shall return a rejection with reason to:
    ({label destination response message ; })*
    (label destination response message )
```

```
destination := phrase | "" phrase "" | '<' phrase '>
response message := '<' phrase '>
```

Example

1) As specified by the requirements described below, the some_system shall return a rejection with reason to:
   a) Operator <this rejection message>;
   b) "Remote Operator" <that rejection message>.

6.6.4.3 Validation Checking

The response declaration fails validation checking if:
1. the group_response_name, [rejection with reason], is already declared
2. a destination, destination_message pair is already declared (i.e., there are duplicate components in the list of a group_response_name)
3. a response_message is used that is not declared
4. a response_message is not of type "<response message>"
5. a destination is used that is not declared
6. a destination is not of type "<destination>"

6.6.5 Acceptance Response

6.6.5.1 Synopsis

The acceptance response introduces a unique group_response_name declaration, "[acceptance ]". The group_response_name is the name for a list of one or more, unique destination, type_name pairs. The purpose of the acceptance response is to provide feedback to the user.

6.6.5.2 Concrete Syntax

The acceptance response must match the following BNF rules:

```
acceptance response :=
```
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

label As specified by the requirements described below, the system_name shall return an [ acceptance ] to ‘:
  ( { label destination response message ‘;’ }*)
  (label destination response message ‘.’

destination : phrase | “phrase” | ‘<’ phrase ‘>’
response message : ‘<’ phrase ‘>’

Example

1) As specified by the requirements described below, the some_system shall return an [acceptance] to:
   a) Operator <this acceptance message>;
   b) “Remote Operator” <that acceptance message>.

6.6.5.3 Validation Checking

   The response declaration fails validation checking if:
   1. the group_response_name, [acceptance], is already declared
   2. a destination, response_message pair is already declared in a group_response_name (i.e., there are duplicate components in the list of a group_response_name)
   3. a response_message is used that is not declared
   4. a response_message is not of type “<response message>”
   5. a destination is used that is not declared
   6. a destination is not of type “<destination>”.

6.6.6 Send Response

6.6.6.1 Synopsis

   The send response introduces a unique group_response_name declaration. The group_response_name is the name for a list of one or more, unique destination, message pairs. The destination message response is an externally visible, testable response. The message of an externally visible response must be of type <response message>. For each externally visible destination, response message pair, zero or more internal send signals may be declared to trigger processing that is described in another specification unit. The internal send signal is not a testable requirement. The message of an internal send signal does not have to be of type <response message>.

6.6.6.2 Concrete Syntax

   A send response must match the following BNF rules:
   send response :=
      label As specified by the requirements described below, the system_name shall send
      a [‘ [ group_response_name ]’] to ‘:
      ({{ label destination response message ‘;’ })
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

\{ \text{label internal send signal of a signal message ';'} \} \ast \} \ast \}
\text{( label destination response message ';')}
\{ \text{label internal send signal of a signal message ';'} \} \ast \}
\text{label internal send signal of a signal message '.)}
\mid \ ( \text{label destination response message '.') }

\text{group response name := phrase}
\text{destination := phrase} \mid \"\text{phrase}\" \mid \text{'<'} \text{phrase} \text{>'}
\text{response message := '<'} \text{phrase} \text{>'}
\text{signal message := '<'} \text{phrase} \text{>'}

Example

1) As specified by the requirements described below, the some_system shall send a [did something response] to:
   a) Operator <this to send>;
   b) internal send signal of a <this send signal>;
   c) "Remote Operator" <that to send>;
   d) Printer <the other thing to print>.

6.6.6.3 Validation Checking

The send response fails validation checking if:
1. the group_response_name is already declared
2. a destination, response_message pair is already declared in a group_response_name (i.e.,
   there are duplicate components in the list of a group_response_name)
3. a destination, signal_message pair is already declared in the list of internal send signals fol­
   lowing an external send response (i.e., there are duplicate components in the list of internal
   send signals)
4. a response_message is used that is not declared
5. a response_message is not of type "<response message>"
6. a signal_message is used that is not declared
7. a destination is used that is not declared
8. a destination is not of type "<destination>"

6.6.7 Commit Response

6.6.7.1 Synopsis

The commit response declaration introduces a group_response_name for one or more
messages that are committed. A committed message is considered "safe" data, i.e., saved, if the
system fails. Each message that is committed may also have zero or more internal commit
signals, which are used to trigger processing in another specification unit. An internal commit
signal is not a testable requirement.
6.6.7.2 Concrete Syntax

A commit response must match the following BNF rules:

\[
\text{commit response} := \\
\hspace{1em} \text{label} \hspace{0.5em} \text{As specified by the requirements described below ', the system_name shall commit a [' group response name '] as :} \\
\hspace{2em} ( \{ ( \text{label message}_1 \{ \text{to message}_2 \}' \} ) \\
\hspace{3em} \{ ( \text{label internal commit signal of a message}_1 ' ;' ) \} * \} *) \\
\hspace{4em} ( \text{label message}_1 \{ \text{to message}_2 \} ' ;' ) \\
\hspace{5em} (\{ ( \text{label internal commit signal of a message}_1 ' ;' ) \} * \} *) \\
\hspace{6em} ( \text{label internal commit signal of a message}_1 '.' ) \\
\hspace{7em} ( \{ ( \text{label message}_1 \{ \text{to message}_2 \} '.' \} ) \\
\hspace{8em} | ( \text{label message}_1 \{ \text{to message}_2 \} '.' ) \\
\text{group response name} := \text{phrase} \\
\text{message}_1 := '<' \text{phrase>}' \\
\text{message}_2 := '<' \text{phrase>}'
\]

Example

4) As specified by the requirements described below, the some_system shall commit a [did that commit response] as:
   a) <that data to commit>;
   b) internal commit signal of a <committed data signal >;
   c) internal commit signal of a <committed data signal as well>;
   d) <part of the data> to <a larger part of the data>.

6.6.7.3 Validation Checking

The response declaration fails validation checking if:
1. a message$_1$ is not declared
2. a message$_2$ is not declared
3. a unique reference does not exist from message$_2$ to message$_1$
4. the group_response_name is already declared
5. a destination, message pair is already declared in a group_response_name (i.e., there are duplicate components in the list of a group_response_name)
6. a destination, signal_message pair is already declared in the list of internal commit signals following a commit response (i.e., there are duplicate components in the list of internal commit signals)

6.6.8 Update Response

6.6.8.1 Synopsis

The update response declaration introduces a local_response_name for one or more
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

Type names that are updated to a type name. An updated message is not considered "safe", i.e., saved, if the system fails. Each data type name that is updated may also have zero or more internal update events which are used to trigger processing in another specification unit. An internal update signal is not a testable requirement.

6.6.8.2 Concrete Syntax

An update response must match the following BNF rules:

```
update response ::= 
  label As specified by the requirements described below, the system_name shall update a '[ group_response_name ]' as:
  ( { ( label message_1 { to message_2 ; } )
    ( ( label internal update signal of a message_1 ; ')* } )* )
  ( label message_1 { to message_2 ; } ; )
  ( label internal update signal of a message_1 . ' )
  | ( label message_1 { to message_2 } . ' )
```

```
group_response_name ::= phrase
message_1 ::= '<' phrase '>'
message_2 ::= '<' phrase '>
```

Example

5) As specified by the requirements described below, the some_system shall update a [did the other update response] as:
   a) <the other data to update>;
   b) <part of the data to update> to <a bigger part of the data>;
   c) <even more data to update>;
   d) internal update signal <even more data updated signal>.

6.6.8.3 Validation Checking

The update response fails validation checking if:
1. a message_1 is not declared
2. a message_2 is not declared
3. a unique reference does not exist from message_2 to message_1
4. the group_response_name is already declared
5. the destination, message, message triple is already declared in a group_response_name (i.e., there are duplicate components in the list of a group_response_name)
6. a destination, signal_message pair is already declared in the list of internal update signals following an update response (i.e., there are duplicate components in the list of internal update signals)
6.6.9 Signal Response

6.6.9.1 Synopsis

The signal response declaration introduces a group signal name for one or more, unique destination, message pairs. The destination is an “internal signal”. The signal is used to couple processing described in different specification units. The signals are not associated with an externally visible send, commit, or update response action. Signals are not testable requirements.

6.6.9.2 Syntax

A signal response must match the following BNF rules:

```
signal response :=
  label As specified by the requirements described below ',' the system_name shall
  signal a [ group_response_name ']' to ':'
  { label internal signal of a signal message ':' }*
  label internal signal of a signal message ':'
```

```
group_response_name := phrase
signal message := '<' phrase '>'
```

Example

6) As specified by the requirements described below, the some_system shall signal a [signalling some response] to:
   a) internal signal of a <something happened signal>.

6.6.9.3 Validation Checking

The response declaration fails validation checking if:
1. a message is not declared
2. the group_response_name is already declared
3. a destination, message pair is already declared in a group_response_name (i.e., there are duplicate components in the list of a group_response_name)

6.7 Stimulus Response Matching Section

6.7.1 Synopsis

The stimulus response matching section allows the user to match a stimulus in a specific group stimulus name with a response in a specific group response name. This allows the user to override the stimulus response matching rules.
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

6.7.2 Concrete Syntax

A Stimulus Response Matching Section must match the following BNF rules:

\[
\text{Stimulus Response Matching Section :=}
\]
\[
\text{Stimulus Response Matching `::`}
\]
\[
\{ \text{label \ The \ source `< message_1 '> in `[ group stimulus name ']` is only matched with destination `< message_2 '> in `[ group response name ']` . } \}
\]

\[
group stimulus name := \text{phrase}
\]
\[
group response name := \text{phrase}
\]
\[
source := \text{phrase} | `" phrase "" | `< phrase `>
\]
\[
destination := \text{phrase} | `" phrase "" | `< phrase `>
\]
\[
message_1 := `< phrase `>
\]
\[
message_2 := `< phrase `>
\]

6.7.3 Validation Checking

The stimulus response matching section fails validation checking if:

1. the source is not of type <source>
2. the destination is not of type <destination>
3. the message_1 is not of type <request message>
4. the message_2 is not of type <response message>
5. the group_stimulus_name is not declared
6. the group_response_name is not declared
7. the source, message_1 pair is not in the group_stimulus_name
8. the destination, message_2 pair is not in the group_response_name
9.

6.8 Requirements Section

6.8.4 General

6.8.4.1 Synopsis

The requirement section describes the processing required to transform a stimulus into a response. The requirements section is composed of one or more requirement statements. The order of the requirements is rejection processing, acceptance processing, followed by stimulus response requirements.

6.8.4.2 Concrete Syntax

A requirements section must match the following BNF rules:

\[
\text{Requirements Section :=}
\]
\[
( \{ \text{rejection requirement} \} + \{ \text{acceptance requirement} \} * \{ \text{stimulus response requirement} \} * )
\]
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

\[
((\text{acceptance requirement}) + (\text{stimulus response requirement})^*)
\]

| \{stimulus response requirement\} +

Example

Requirements:

1) Upon receipt of a [do this request], if {there is some bad data} the some_system shall return a [rejection with reason].

2) Upon receipt of a [do the other thing request], the some_system shall return a [acceptance] to "Remote Operator".

3) Upon receipt of a [do that request] from "internal signal", the some_system shall send a [did something response].

4) Upon receipt of a [do this request], if {some condition holds}, the some_system shall commit a [did that commit response].

5) Upon receipt of a [do this request], the some_system shall update a [did the other update response], if {some other condition holds}.

6) Upon receipt of a [do that request], if {some interesting condition holds}, the some_system shall:
   a) signal a [signalling some response];
   b) send a [did something response] to "Remote Operator";
   c) commit a [did that commit response].

7) Upon receipt of a [do this request], the some_system shall signal a [signalling some response].

8) If {some condition holds}, the some_system shall send a [did something response] to Operator.

6.8.4.3 Validation Checking

The requirement section fails validation checking if:
1. a rejection requirement, if present, fails
2. an acceptance requirement, if present, fails
3. a stimulus response requirement, if present, fails

6.8.5 Rejection Requirement

6.8.5.1 Synopsis

The rejection requirement describes the generation of a rejection with reason.
6.8.5.2 Syntax

rejection requirement :=
   label Upon receipt of a [ group response name ] { from source } ,
   { if { logical condition } } , the system_name shall return a
   [ rejection with reason ] { to destination } .

Example
1) Upon receipt of a [do this request], if {some check has failed} the some_system shall return
   a [rejection with reason].

6.8.5.3 Validation Checking

A rejection requirement fails validation checking if:
1. the group_response_name, if present, is not declared
2. the source, if present, is not declared
3. the source, if present, is not of type "<source>"
4. the source, if present, is not declared as a source in the group_response_name
5. the destination, if present, is not declared
6. the destination, if present, is not of type "<destination>"
7. the destination, if present, is not declared as a destination in the rejection with reason
   group_response_name
8. the logical_condition, if present, does not evaluate to type <bool> (Section 8)

6.8.6 Acceptance Requirement

6.8.6.1 Synopsis

The acceptance requirement describes the generation of an acceptance response.

6.8.6.2 Concrete Syntax

acceptance requirement :=
   label Upon receipt of a [ group response name ] { from source } ,
   { if { logical condition } } , the system_name shall return an
   [ acceptance ] { to destination } .

Example

214
2) Upon receipt of a [do the other thing request], the some_system shall return an [acceptance] to “Remote Operator”.

6.8.6.3 Validation Checking

An acceptance requirement fails validation checking if:
1. the group_response_name is not declared
2. the destination, if present, is not declared
3. the destination, if present, is not of type “<destination>”

6.8.7 Stimulus Response Requirement

6.8.7.1 Synopsis

A stimulus response requirement expresses either a direct relationship between a stimulus and one or more responses or the generation of one or more responses (the stimulus is optional). The stimulus response requirement can be extended by describing response qualification and/or response response requirements. A response qualification is a predicate that is applied to an attribute of the response message or a boolean condition. A response response requirement is a requirement that cascades from the previous stimulus response requirement. The evaluation rules for response qualification and response response requirements are in Chapter 4.

6.8.7.2 Concrete Syntax

A stimulus response requirement must match the following BNF rules:

```
stimulus response requirement :=
  (label stimulus part { if '{' logical condition '}'} the system_name shall action '.')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')
| (label stimulus part the system_name shall action '{' if '{' logical condition '}'} ')')

stimulus part :=
  Upon receipt of a '[' group stimulus name ']' { from source } ';

action :=
  send a '[' group response name ']' { to destination }
  '{' '{' send response qualification '}' '}' '{' commit response response '}'
  '{' update response qualification '}'
```
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

\[
\begin{align*}
\text{action list :=} \\
\{ \text{label send a [ group response name [ signal response response ] ]} \\
\{ . ‘{ signal response qualification ‘} ‘} ‘\} ‘* ‘. signal response response ‘
\} ‘;’ \\
\text{label commit a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ commit response qualification ‘} ‘} ‘* ‘. commit response response ‘
\} ‘;’ \\
\text{label update a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ update response qualification ‘} ‘} ‘* ‘. update response response ‘
\} ‘;’ \\
\text{label signal a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ signal response qualification ‘} ‘} ‘* ‘. signal response response ‘ ‘;‘
\} ‘;’ \\
\{ \text{label send a [ group response name [ signal response response ] ]} \\
\{ . ‘{ signal response qualification ‘} ‘} ‘* ‘. send response response ‘
\} ‘;’ \\
\text{label commit a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ commit response qualification ‘} ‘} ‘* ‘. commit response response ‘
\} ‘;’ \\
\text{label update a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ update response qualification ‘} ‘} ‘* ‘. update response response ‘
\} ‘;’ \\
\text{label signal a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ signal response qualification ‘} ‘} ‘* ‘. signal response response ‘
\} ‘;’ \\
\{ \text{label send a [ group response name [ signal response response ] ]} \\
\{ . ‘{ signal response qualification ‘} ‘} ‘* ‘. send response response ‘
\} ‘;’ \\
\text{label commit a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ commit response qualification ‘} ‘} ‘* ‘. commit response response ‘
\} ‘;’ \\
\text{label update a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ update response qualification ‘} ‘} ‘* ‘. update response response ‘
\} ‘;’ \\
\text{label signal a [ group response name [ signal response response ] ]} \\
\{ ‘. ‘{ signal response qualification ‘} ‘} ‘* ‘. signal response response ‘
\} ‘;’ \\
\end{align*}
\]

\[
\begin{align*}
\text{group response name := phrase} \\
\text{source := phrase | ‘“ phrase ‘”‘ | ‘< phrase ‘>‘} \\
\text{destination := phrase | ‘“ phrase ‘”‘ | ‘< phrase ‘>‘}
\end{align*}
\]

Example

3) Upon receipt of a [do that request] from “internal signal”, the some_system shall send a [did something response].

4) Upon receipt of a [do this request] , if \{some condition holds\}, the some_system shall commit a [did that commit response].

5) Upon receipt of a [do this request] , the some_system shall update a [did the other update response], if \{some other condition holds\}.

6) Upon receipt of a [do that request] , if \{some interesting condition holds\}, the some_system shall:
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

a) signal a [signalling some response];
b) send a [did something response];
c) commit a [did that commit response].

7) Upon receipt of a [do this request], the some_system shall signal a [signalling some response].

8) If {some condition holds}, the some_system shall send a [did something response] to Operator.

6.8.7.3 Validation Checking

The stimulus response requirement fails validation checking if:
1. the stimulus part, if present, fails
2. the condition part, if present, fails (Section 8)
3. an action, if present, fails
4. an action list, if present fails.

A stimulus part fails validation checking if:
1. the group_stimulus_name is not declared
2. the source, if present, is not declared
3. the source, if present, is not of type "<source>"
4. the source, if present, is not in the group_stimulus_name

A condition part fails validation checking if:
1. the logical condition does not evaluate to a boolean type (Section 8)

An action fails validation checking if:
1. the group_response_name is not declared
2. the destination, if present, is not declared
3. the destination, if present, is not of type "<destination>"
4. the destination, if present, is not in the group_response_name
5. a send response response, if present, fails
6. a send response response list, if present, fails
7. a commit response qualification, if present, fails
8. a commit response response, if present fails
9. a commit response response list, if present fails
10. an update response qualification, if present, fails
11. an update response response, if present, fails
12. an update response response list, if present, fails
13. a signal response qualification, if present, fails
14. a signal response response, if present, fails
15. a signal response response list, if present, fails

An action list fails validation checking if:
1. the group_response_name is not declared
2. the destination, if present, is not declared
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

3. the destination, if present, is not of type "<destination>"
4. the destination, if present, is not in the group_response_name
5. a send response response, if present, fails
6. a commit response qualification, if present, fails
7. a commit response response, if present fails
8. an update response qualification, if present, fails
9. an update response response, if present, fails
10. a signal response qualification, if present, fails
11. a signal response response, if present, fails

6.8.8 Response Qualification Requirement

6.8.8.1 Synopsis

A response qualification requirement is either a predicate or a boolean condition. If it is a predicate, then it is applied to an attribute of the destination message or the complete destination message of a sent, committed, updated, or signalled group response name. The destination name may also be applied to a predicate for a sent group response name.

6.8.8.2 Concrete Syntax

The send, update, commit, and signal response qualifications must match the following BNF rules:

```
send response qualification :
  ( '{' send attribute '}' predicate body text { send parameter predicate body text })*
  { send parameter }
| ( { predicate body text send parameter }* (predicate body text '{' send attribute '}')
  { predicate body text send parameter }* {predicate body text} )
| '{' boolean condition '}'
```

```
send attribute :=
  ( ( all of) |(one or more of )| (none of) the '{' message '}' ( of | specified in
  | associated with ) the sent '{' [' group response name ']' '}' )
| ( ( All of) |(One or more of )| (None of) the '{' message '}' ( of | specified in
  | associated with ) the sent '{' [' group response name ']' '}' )
| the '{' message '}' ( of | specified in | associated with ) the sent
  '{' [' group response name ']' '}' )
| The '{' message '}' ( of | specified in | associated with ) the sent
  '{' [' group response name ']' '}' )
| ( the sent '{' [' group response name ']' '}' )
| ( The sent '{' [' group response name ']' '}' )
| ( the '{' destination '}' of the sent '{' [' group response name ']' '}' )
| ( The '{' destination '}' of the sent '{' [' group response name ']' '}' )
```

```
group response name:= phrase
message:= phrase | "" phrase "" | '<' phrase '>' | set of '<' phrase '>'
```
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

destination := phrase | ‘“’ phrase ‘”’ | ‘<’ phrase ‘>’
predicate body text := phrase
send parameter := data expression
boolean condition := phrase | ‘<’ phrase ‘>’

Examples

{ The {colour field} specified in the sent { [did something response] } is set to blue }

{ some boolean condition holds }

custom response qualification :=

( {’ commit attribute ‘} predicate body text { commit parameter predicate body text }* { commit parameter })
| ( predicate body text commit parameter )* ( predicate body text

‘{’ commit attribute ‘}’ ) ( { predicate body text commit parameter }* { predicate body text })
| {’ boolean condition ‘}

custom attribute :=

( ( all of)(one or more of)(none of) the {’ message ‘} ( of | specified in | associated with ) the committed {’ {’ group response name ‘} ‘} )
| ( ( All of)(One or more of)(None of) the {’ message ‘} ( of | specified in | associated with ) the committed {’ {’ group response name ‘} ‘} )
| ( the {’ message ‘} ( of | specified in | associated with ) the committed ‘{’ {’ group response name ‘} ‘} )
| ( The {’ message ‘} ( of | specified in | associated with ) the committed ‘{’ {’ group response name ‘} ‘} )
| ( the committed ‘{’ {’ group response name ‘} ‘} )
| ( The committed ‘{’ {’ group response name ‘} ‘} )

group response name:= phrase
message:= phrase | ‘“’ phrase ‘”’ | ‘<’ phrase ‘>’ | set of ‘<’ phrase ‘>’
predicate body text := phrase
commit parameter := data expression
boolean condition := phrase | ‘<’ phrase ‘>’

{ The committed {{did something response} } is a parameter of some predicate }

{ <bool> }

update response qualification :=

( {’ update attribute ‘} predicate body text { update parameter predicate body text }* { update parameter })
| ( predicate body text update parameter )* ( predicate body text ‘

219
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

update attribute :=
  (( all of) |(one or more of ) |(none of)} the `{ message '} ( of | specified in  
| associated with ) the updated `{ group response name '} 
| ( All of ) |(One or more of ) |(None of ) the `{ message '} ( of | specified in  
| associated with ) the updated `{ group response name '} 
| ( the `{ message '} ( of | specified in | associated with ) } the updated  
`{ group response name '} 
| ( The `{ message '} ( of | specified in | associated with ) } the updated  
`{ group response name '} 
| ( the updated `{ group response name '} 
| ( The updated `{ group response name '} 

group response name:= phrase
message:= phrase | "" phrase "" | '<' phrase '>' | set of '<' phrase '>'
predicate body text := phrase
update parameter := data expression
boolean condition := phrase | '<' phrase '>'

Examples

{ There is a predicate that has {the updated [did the other update response]} as a parameter}. 

{All of the {colour field} associated with the updated [did the other update response] are set to pink}. 

signal response qualification :=
  (`{ signal attribute '} predicat body text } {signal parameter  
  predicate body text })* { signal parameter  }
| {{ predicate body text signal parameter }* (predicate body text `{ signal attribute '} } 
| ( { predicate body text signal parameter }* { predicate body text } )
| `{ boolean condition '}

signal attribute :=
  (( all of) |(one or more of ) |(none of)} the `{ message '} ( of | specified in  
| associated with ) the signaled `{ group response name '} 
| ( All of ) |(One or more of ) |(None of ) the `{ message '} ( of | specified in  
| associated with ) the signaled `{ group response name '} 
| ( the `{ message '} ( of | specified in | associated with ) } the signaled  
`{ group response name '} 
| ( The `{ message '} ( of | specified in | associated with ) } the signaled  
`{ group response name '} 
| ( the signaled `{ group response name '} 

220
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

\[
\begin{align*}
| ( \text{The signaled } \{ \{' \text{group response name}' \} \}' ) \]
\end{align*}
\]

\[\begin{align*}
group \text{ response name} & := \text{phrase} \\
message & := \text{phrase} \mid \text{"" phrase ""} \mid \text{"" phrase ""} \mid \text{"" phrase ""} \\
\text{predicate body text} & := \text{phrase} \\
\text{signal parameter} & := \text{data expression} \\
\text{boolean condition} & := \text{phrase} \mid \text{"" phrase ""} \\
\end{align*}\]

Examples

\{
There is a predicate that has \{all of the \{colour field\} of the signaled \{signalling some response\}\} as the first parameter and \{some other data of interest\} as a second parameter.
\}

\{
some other fascinating condition holds
\}

6.8.8.3 Validation Checking

The response qualification will fail validation checking if:

1. there is not a unique reference from the group_response_name to the message, if present
2. a declared predicate does not match the predicate application, if present
3. the group_response_name used in the response qualification does not match the group_response_name in the stimulus response requirement it is qualifying
4. the group_response_name used in the response qualification does not match the category of the declaration of the group_response_name in the responses section e.g., send a \[x1\] in responses section. When \[x1\] is used in a requirement, it must be for a send stimulus response requirement.
5. the boolean_condition does not evaluate to type \text{<bool>} (Section 8)

6.8.9 Response Response Requirements

6.8.9.1 Synopsis

A response response requirement indirectly expresses a relationship between a stimulus and a response in terms of an "in between" response. The response response requirements are either single requirements or a list of requirements.

6.8.9.2 Concrete Syntax

The send, commit, update, and signal response response requirements must match the following BNF rules:

\[
\begin{align*}
\text{send response response} & := \\
( \text{If a } \{' \text{group response name}' \} \text{is sent } \{ \text{and } \{' \text{logical condition}' \} \} \text{, the system name shall action} ) \\
\end{align*}\]

commit response response :=

221
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

(If a [' group response name ' ] is committed { and '{ logical condition ' } } ;'
the system_name shall action )

update response response :=
(If a [' group response name ' ] is updated { and '{ logical condition ' } } ;'
the system_name shall action )

signal response response :=
( If a [' group response name ' ] is signaled { and '{ logical condition ' } } ;'
the system_name shall action )

The send, commit, update, and signal response response requirement lists must match the following BNF rules:

send response response list :=
If a [' group response name ' ] is sent { and '{ logical condition ' } } ;'
the system_name shall ':' { label action ';' } + label action

commit response response list :=
If a [' group response name ' ] is committed { and '{ logical condition ' } } ;'
the system_name shall ':' { label action ';' } + label action

update response response list :=
If a [' group response name ' ] is updated { and '{ logical condition ' } } ;'
the system_name shall ':' { label action ';' } + label action

signal response response list :=
If a [' group response name ' ] is signaled { and '{ logical condition ' } } ;'
the system_name shall ':' { label action ';' } + label action

action :=
send a [' group response name ' ] { to destination } { '.' send response qualification }*
{ '.' send response response } )
| commit a [' group response name ' ] { '.' commit response qualification }*
{ '.' commit response response }
| update a [' group response name ' ] { '.' update response qualification }*
{ '.' update response response }
| signal a [' group response name ' ] { '.' signal response qualification }*
{ '.' signal response response }

group response name := phrase
destination := phrase | "" phrase "" | '<' phrase '>'
6.8.9.3 Validation Checking

A response response requirement fails validation checking if:
1. a group_response_name is not declared
2. the logical_condition fails to evaluate to a boolean expression (Section 8)
3. the destination, if present, is not declared
4. the destination, if present, is not of type “<destination>”

6.9 Performance Section

6.9.1 Synopsis

The performance section describes the performance requirements for the specification unit. The style for describing the performance requirements is particular to a project.

6.9.2 Concrete Syntax

The performance section must match the following BNF rules:

\[
\text{Performance} \colon \ :
\{ \text{label logical condition} \colon \ . \}^* 
\]

6.9.3 Validation Checking

The performance section fails validation checking if any of the logical conditions fail (Section 8).

6.10 Declarations

6.10.1 General

6.10.1.1 Synopsis

The declarations section introduces data types, data subtypes, data objects (constants, variable system parameters, system state components), stimulus response components, functions, and predicates. The functions and predicates are categorized as local (declared in this specification unit), external (declared in another specification unit), or exported (declared in this specification unit for use in other specification units).

6.10.1.2 Concrete Syntax

A declarations section must match the following BNF rules:

\[
\text{Declarations Section} := \\
\text{Declarations:} \\
\text{type names subsection} \\
\text{subtypes subsection} \\
\text{static associations subsection} 
\]
constant declarations subsection
variable system parameters subsection
system state components subsection
stimulus response components subsection
local functions subsection
local predicates subsection
external functions subsection
external predicates subsection
exported functions subsection
exported predicates subsection

6.10.1.3 Validation Checking

The declaration section fails validation checking if any of its subsections fail.

6.10.2 Type Names Subsection

6.10.2.1 Synopsis

The type names subsection introduces zero or more data types for use. The data types may be a basic type or a set of a basic type.

6.10.2.2 Concrete Syntax

A type names subsection must match the following BNF rules:

\[
\text{type names subsection := }
\]

\[
\text{Type Names:}
\]

\[
\begin{align*}
\{ & \text{ label ':' type name '.'} \\
\| & \text{ label ':' set of type name '.'} \}
\end{align*}
\]

\[
\text{type name := '<' phrase '>'}
\]

\[
\text{set of type name := set of '<' phrase '>'}
\]

6.10.2.3 Validation Checking

A type name declaration fails if:
1. the type name is already declared.
2. the set_of_type_name is already declared.

6.10.3 Subtype Subsection

6.10.3.1 Synopsis

The subtype subsection introduces zero or more subtypes for use. The subtype names introduced may be subtypes of a basic type or subtypes of a set of basic types.
6.10.4 Concrete Syntax

A subtype subsection must match the following BNF rules:

```
subtype subsection :=
  Subtypes:
  { label A subtype name is a subtype of type name '.'
    | label A set subtype name is a subtype of set type name '.' }*
```

```
subtype name := '<' phrase '>
```

```
type name := '<' phrase '>
```

```
set subtype name := set of '<' phrase '>
```

```
s set type name := set of '<' phrase '>
```

6.10.4.1 Validation Checking

The subtype fails validation checking if:
1. the subtype_name is already declared for another type name
2. the subtype_name is already declared for the type_name
3. the type_name is not declared
1. the set_subtype_name is already declared for another set type_name
2. the set_subtype_name is already declared for the set_type_name
3. the set_type_name is not declared

6.10.5 Static Associations Subsection

6.10.5.1 Synopsis

The static associations subsection introduces relationships between types of data objects. The “one or more” and “zero or more” cardinalities describe an association from the type_name_1 or set_type_name_1 to a set of type_name_2. The “exactly one” and “zero or one” cardinalities describe an association from the type_name_1 or set_type_name_1 to the type_name_2. The static associations are templated, local function or predicate declarations.

6.10.5.2 Concrete Syntax

A static association subsection must match the following BNF rules:

```
static association subsection :=
  Static Associations:
  { label Every type name_1 is associated with : { label association ';'} }+
    label association '.'
  | label Every set type name_1 is associated with : { label association ';'} }+
    label association '.'
  | label Every type name_1 is associated with association '.' }*
  | label Every set type name_1 is associated with association '.' }*
```
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

association :=
  zero or one type name₂
| zero or more type name₂
| one or more type name₂
| exactly one type name₂

| type name₁ := '<' phrase '>'
| type name₂ := '<' phrase '>

6.10.5.3 Validation Checking

The static association declaration fails validation checking if:
1. a type name₁ is not declared.
2. a type name₂ is not declared.
3. a set type name₁ is not declared.
4. a set type name₂ is not declared.
5. a static association is already declared for type name₁ or its subtypes and type name₂ or its subtypes.
6. a static association is already declared for set type name₁ or its subtypes and type name₂ or its subtypes.
7. type name₁ is declared in an "exactly one" or a "one or more" association with a type name₂ that has the same as type name₁.
8. set type name₁ is declared in a "one or more" or a "zero or more" association with a type name₂ that has the same base type as set type name₁.

6.10.6 Constant Declarations subsection

6.10.6.1 Synopsis

The constant declaration subsection is composed of a list of zero or more constants. The constant declaration subsection introduces data objects of a specific type.

6.10.6.2 Concrete Syntax

A constant declarations subsection must match the following BNF rules:
constant declarations subsection :=
  Constants ':'
  (label constant name ': ' type name '.' )*

constant name := phrase | "" phrase "" | '<' phrase '>
| type name := '<' phrase '>'
6.10.6.3 Validation Checking

The constant declaration fails if:
1. the type_name is not declared
2. the constant_name is already declared for the type_name in the constant, VSP, system state, or stimulus response component sections

6.10.7 Variable System Parameters Subsection

6.10.7.1 Synopsis

The variable system parameter subsection is composed of a list of zero or more variable system parameters. The variable system parameters are data objects of a specific type.

6.10.7.2 Concrete Syntax

A variable system parameters subsection must match the following BNF rules:
variable system parameters subsection :=
    Variable System Parameters:
        {label variable system parameter name ':' type name '}' *

variable system parameter name := phrase | "" phrase "" | '<' phrase '>'

6.10.7.3 Validation Checking

The variable system parameter declaration fails if:
1. the type_name is not declared.
2. the variable system parameter is already declared in the constant, VSP, system state, or stimulus response component subsection.

6.10.8 System State Components Subsection

6.10.8.1 Synopsis

The system state component subsection is composed of a list of zero or more system state components. The system state components are data objects of a specific type.

6.10.8.2 Concrete Syntax

A system state component subsection must match the following BNF rules:
system state component subsection :=
    System State Components ':
        { label system state component name ':' { type name '}' }*

system state component name := phrase | "" phrase "" | '<' phrase '>'
type name := '='<phrase '>>' | set of '='<phrase '>>'

6.10.8.3 Validation Checking

The system state component declaration fails if:
1. the type_name is not declared
2. the system state component is already declared with the type_name in the constant, VSP, system state, or stimulus response component section

6.10.9 Stimulus Response Components Subsection

6.10.9.1 Synopsis

The stimulus response components subsection is composed of a list of zero or more components. The components may be source names, destination names, request messages, or response messages. The stimulus response components sections introduces the stimuli and responses used in the specification unit in one place. The request_message is declared as a subtype of <request message>. The response_message is declared as a subtype of <response message>. The source is declared as a subtype of <source>. The destination is declared as a subtype of <destination>.

6.10.9.2 Syntax

A stimulus response component subsection must match the following BNF rules:

stimulus response component subsection :=
    Stimulus Response Components ':'
    { label source is a <source> '.'
    | label '<'request message '>' is a <request message> '.'
    | label destination is a <destination> '.'
    | label '<'response message '>' is a <response message> '.' }*

source := phrase | '<phrase '>' | "" phrase ""
destination := phrase | '<phrase '>' | "" phrase ""
request message := phrase
response message := phrase

6.10.9.3 Validation Checking

The stimulus response component declaration fails if:
1. the source is already declared as a <source>
2. the destination is already declared as a <destination>
3. the request_message is already declared as a type name or a subtype name
4. the response_message is already declared as a type name or a subtype name
5. the request_message is already declared
6. the response_message is already declared
6.10.10  Local Functions Subsection

6.10.10.1 Synopsis

The local functions subsection introduces a list of locally declared functions for use.

6.10.10.2 Concrete Syntax

A local functions subsection must match the following BNF rules:

\[
\text{local functions subsection :=} \\
\quad \text{Local Functions:} \\
\quad \quad \{ \text{label function is a function ':' return type name '.*} \}\ *
\]

\[
\text{function:=} \\
\quad \{ \{ \text{parameter } \}+ \text{function body text } \{ \text{parameter } \} * \}+ \\
\quad | \{ \{ \text{function body text} \} \{ \text{parameter } \} + \} + \{ \text{function body text} \}
\]

\[
\text{return type name:= ' '< phrase '>' } | \text{set of ' '< phrase '>'}
\]

\[
\text{parameter := '(' ' '< phrase '>' ')'} | \text{set of ' '< phrase '>' ' )'}
\]

\[
\text{function body text : = } \{ \text{phrase } \}+
\]

6.10.10.3 Validation Checking

The local function subsection fails typechecking if:
1. the return_type_name is not declared.
2. the return_type_name is of type bool (this is a predicate)
3. the function is already declared.
4. a parameter is not declared as a data type

6.10.11  Local Predicates Subsection

6.10.11.1 Synopsis

The local predicates subsection introduces locally declared predicates for use.

6.10.11.2 Concrete Syntax

A local predicates subsection must match the following BNF rules:

\[
\text{local predicate subsection :=} \\
\quad \text{Local Predicates:} \\
\quad \quad \{ \text{label predicate is a predicate '.*} \}\ *
\]

\[
\text{predicate:=} \\
\quad \{ \{ \text{parameter } \}+ \text{predicate body text } \{ \text{parameter } \} * \}+ \\
\quad | \{ \{ \text{predicate body text} \} \{ \text{parameter } \} + \} + \{ \text{predicate body text} \}
\]
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

\[
\text{parameter} := \left( \text{` <` phrase `>' } \right) \mid \left( \text{` set of ` <` phrase `>' } \right)
\]

\[
\text{function body text} := \{ \text{ phrase } \}^+ 
\]

6.10.11.3 Validation Checking

The local predicate declaration subsection fails if:
1. the predicate is already declared
2. a parameter is not declared as a data type

6.10.12 External Functions Subsection

6.10.12.1 Synopsis

The external functions subsection introduces a list of externally declared functions for use.

6.10.12.2 Concrete Syntax

An external functions subsection must match the following BNF rules:

\[
\text{external functions subsection} := \\
\text{External Functions:} \\
\{ \text{label function as specified in the `\' title string `\' : return type name `.' } \}^+ \\
\]

\[
\text{function} := \\
\{ \{ \text{ parameter } \}^+ \text{ function body text } \{ \text{ parameter } \}^+ \}^+ \{ \text{ function body text } \}
\]

\[
\text{return type name} := \left( \text{` <` phrase `>' } \right) \mid \text{ set of ` <` phrase `>' } \\
\text{parameter} := \left( \text{` <` phrase `>' } \right) \mid \left( \text{` set of ` <` phrase `>' } \right)
\]

\[
\text{function body text} := \{ \text{ phrase } \}^+ \\
\text{title string} := \text{ phrase}
\]

6.10.12.3 Validation Checking

The external function subsection fails typechecking if:
1. the return type name is not declared
2. the function is already declared
3. a parameter is not declared as a data type

6.10.13 External Predicates Subsection

6.10.13.1 Synopsis

The external predicates subsection introduces externally declared predicates for use. These predicates are declared in another specification unit and are used in this specification unit.
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

6.10.13.2 Concrete Syntax

An external predicates subsection must match the following BNF rules:

\[
\text{external predicate subsection} :=
\]

\[
\text{External Predicates:}\]

\[
\{ \text{ label predicate as specified in } \text{\'V title string \text{\' \text{'}} } \}
\]

\[
\text{parameter} :=
\]

\[
\{ { \text{ parameter } } { \text{+ predicate body text } { \text{ parameter } } } * } \} +
\]

\[
| \{ \{ \text{ predicate body text } \} \{ \text{ parameter } \} + \} { \text{ predicate body text } \}
\]

\[
\text{parameter} := \text{\(' (' <' phrase '>' ')') | (' set of '<' phrase '>' ')')}
\]

\[
\text{function body text} := \{ \text{ phrase } \} +
\]

\[
\text{title string} := \text{ phrase}
\]

6.10.13.3 Validation Checking

The external predicate declaration subsection fails if:

1. the type name is not declared
2. the predicate is already declared
3. a parameter is not declared as a data type

6.10.14 Exported Functions Subsection

6.10.14.1 Synopsis

The exported functions subsection introduces a list of declared functions that are exported for use.

6.10.14.2 Concrete Syntax

An exported functions subsection must match the following BNF rules:

\[
\text{exported functions subsection} :=
\]

\[
\text{Exported Functions:}\]

\[
\{ \text{ label function as specified in the } \text{\'V title string \text{\' : } return type name \text{\'} } \}
\]

\[
\text{function} :=
\]

\[
\{ { \text{ parameter } } { \text{+ function body text } { \text{ parameter } } } * } \} +
\]

\[
| \{ \{ \text{ function body text } \} \{ \text{ parameter } \} + \} { \text{ function body text } \}
\]

\[
\text{return type name} := \text{\'<' phrase '>' | set of '<' phrase '>'}
\]

\[
\text{parameter} := \text{\(' (' <' phrase '>' ')') | (' set of '<' phrase '>' ')')}
\]

\[
\text{function body text} := \{ \text{ phrase } \} +
\]

\[
\text{title string} := \text{ phrase}
\]
6.10.14.3 Validation Checking

The exported function subsection fails typechecking if:
1. the type_name is not declared
2. the function is already declared
3. a parameter is not declared as a data type

6.10.15 Exported Predicates Subsection

6.10.15.1 Synopsis

The exported predicates subsection introduces predicate declarations for use outside the specification.

6.10.15.2 Concrete Syntax

An exported predicates subsection must match the following BNF rules:

\[
\text{exported predicate subsection :=}
\text{Exported Predicates:}
\{ \text{label predicate as specified in } \text{'\'} \text{ title string } \text{'} \text{'} \}\ast
\]

\[
\text{predicate :=}
\{ \{ \text{parameter } \}+ \text{ predicate body text } \{ \text{parameter } \} \ast \}+ \\
\| \{ \{ \text{predicate body text } \} \{ \text{parameter } \}+ \}+ \{ \text{predicate body text } \}
\]

\[
\text{parameter := '( ' '<' phrase '>' ')'} \mid '( ' \text{set of } '<' \text{ phrase } '>' ')'
\]

\[
\text{function body text : = \{ phrase \}+}
\]

\[
\text{title string := phrase}
\]

6.10.15.3 Validation Checking

The exported predicate declaration subsection fails typechecking if:
1. the type_name is not declared
2. the predicate is already declared
3. a parameter is not declared as a data type

7. Data Expressions

7.1 Synopsis

A data expression denotes a data object or a set of data objects.

7.2 Syntax

A data expression must match the following BNF rules:

\[
\text{data expression :=}
\]
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

<table>
<thead>
<tr>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>the input</td>
</tr>
<tr>
<td>the '{ message '}' of '{ data expression '}'</td>
</tr>
<tr>
<td>the '{ message '}' specified in '{ data expression '}'</td>
</tr>
<tr>
<td>the '{ message '}' associated with '{ data expression '}'</td>
</tr>
<tr>
<td>'{ the message '}'</td>
</tr>
<tr>
<td>function application</td>
</tr>
</tbody>
</table>

function_application :=
| {{ parameter }+ function body text { parameter }* }+ |
| {{ function body text } { parameter }+ }+ { function body text } |
| {{ function body text } { parameter }+ }+ as specified in the '{ title thread '} |

parameter := '{ data expression '}'
function_body_text := { phrase }+
title_thread := phrase
message := phrase | '< phrase '>' | "" phrase "" | set of '< phrase >'

7.3 Validation Checking

The data expression fails if:
1. A message is not a declared data type, subtype, constant, variable system parameter, system state component, or integer
2. A function declaration does not match the function application
3. A message is not uniquely related to the data expression, if present
4. A message is not uniquely related to the input, if the data expression is not present

The rules for validation checking a data_expression are those specified in the DSPEC notation.

8. Logical Conditions

8.1 Synopsis

A logical condition is an expression that denotes a condition that is either true or false.

8.2 Syntax

A logical condition must match the following BNF rules:

logical_condition:=
| all of the following conditions are satisfied : Labelled condition list |
| one or more of the following conditions are satisfied : Labelled condition list |
| not '{ logical condition '} |
| '{ logical condition '} and '{ logical condition '} |
| '{ logical condition '} or '{ logical condition '} |
Appendix A. Definition of the Stimulus Response Requirements Specification Notation

<table>
<thead>
<tr>
<th>'{' predicate application'}'</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_application</td>
</tr>
<tr>
<td>'{' message '}'</td>
</tr>
</tbody>
</table>

// quantified conditions
| one or more of the '{' message '} such that '{' logical_condition '}' |
| all of the '{' message '} such that '{' logical_condition '}' |
| none of the '{' message '} such that '{' logical_condition '}' |

message:= phrase | '<' phrase '>' | "" phrase "" | set of '<' phrase '>' |

Labelled condition list :=
{ label '{' logical_condition '}' ';' }+
label '{' logical_condition '}'

predicate_application :=
{ { parameter }+ predicate_body_text { parameter }* }+
| { { predicate_body_text } { parameter }+ }+ { predicate_body_text }
| { { predicate_body_text } { parameter }+ }+ as specified in the 'V' title thread 'V'

table_application :=
{ { parameter }+ { predicate ';'}* }+
table_name

parameter := '{' data expression '}'
function_body_text := { phrase }+
title_thread := phrase
message := phrase | '<' phrase '>' | "" phrase ""
table_name := phrase

8.3 Validation Checking

The logical condition fails if:
1. A message is not declared of type bool, a subtype of base type bool, constant of type bool or a subtype with base type bool, variable system parameter of type bool or a subtype of base type bool, or system state component of type bool or a subtype of base type bool
2. a predicate declaration does not match a predicate application
3. a table_declaration does not match a table_application
4. the evaluation of a parameter in a predicate_application or a table_application is unambiguous
5. in a quantified expression, the message is not a type or subtype declaration.

The rules for validation checking a logical_condition are those specified in the DSPEC notation.
Appendix B. Definition of the Data Specification Notation

1 Introduction

The data specification (DSPEC) notation is defined in this Appendix. The definition is the requirements specification of the notation and is not intended to constrain the design or implementation of the tool support for the notation.

The structure of this Appendix is as follows. The basic components of the notation are described in Section 2. User commands and error messages are provided in Section 3. The DSPEC notation is introduced in Section 4. The requirements evaluation rules are described in Section 5. The detailed syntax and evaluation rules are provided in Section 6.

2. Basic Components of the DSPEC Notation

The lexical elements, reserved keywords, built-in components, and the underlying concepts of the notation are described in this section.

2.1 Lexical Elements

The lexical elements in DSPEC include comments, labels, punctuation, system_names, and phrases.

• A comment is a string of printable ASCII characters that are delimited by the percent sign character
• A label element is composed one or two ASCII characters followed by a closing parenthesis character
• A number element is an integer
• A punctuation element may be a comma, period, semi-colon, full-colon, opening or closing brace, opening or closing square bracket, opening or closing parenthesis, opening or closing angle bracket, double quote, or the backslash character
• The system_name element is a string of one or more printable ASCII characters that does not include punctuation characters or whitespace characters (tab, space, newline)
• A phrase element is composed of one or more printable ASCII characters that does not contain opening and closing braces, opening and closing square brackets, opening and closing parentheses, opening and closing angle brackets, the percent sign character, or the backslash character. A phrase that is delineated by angle brackets, square brackets, backslashes, or double quotes may contain whitespace characters. One or more consecutive whitespace characters in a delineated phrase are scanned as a single occurrence of the space character. Whitespace at the beginning and the end of a phrase (delineated or not) are removed

The number of characters in a phrase, system_name, or comment must not exceed 1024 or include the EOF.
2.2 Reserved Keywords and Characters

The reserved keywords and characters in DSPEC are described in Table 1:

<table>
<thead>
<tr>
<th>Reserved Keywords</th>
<th>Reserved Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>A from such there</td>
<td>. . &lt; }</td>
</tr>
<tr>
<td>a is that to</td>
<td>( &gt; % )</td>
</tr>
<tr>
<td>An not the with</td>
<td>an null The</td>
</tr>
<tr>
<td>associated of There</td>
<td>associated of There</td>
</tr>
</tbody>
</table>

Table 1: Reserved Keywords and Characters in DSPEC

2.3 Built-in Components

The built-in types, sources, destinations, and constants in DSPEC are described in Table 2:

<table>
<thead>
<tr>
<th>Built-in Type Names</th>
<th>Built-in Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bool&gt;</td>
<td>TRUE :&lt;bool&gt;</td>
</tr>
<tr>
<td>&lt;date&gt;</td>
<td>FALSE:&lt;bool&gt;</td>
</tr>
<tr>
<td>&lt;centimetre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;millisecond&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;decimetre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;year&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;second&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;millilitre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;month&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;minute&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;decilitre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day of year&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;hour&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;litre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day of month&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;metre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;celsius&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day of week&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;kilometre&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;day&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;millimetre&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Built-in Components of DSPEC
2.4 The DSPEC context

The context includes:

- data types. A data type is the name for a set of objects. For example, "integer" is the name of the data type that contains all the integers.
- data subtypes. A subtype is the name for a subset of a data type. For example, "positive integers" is the name of a subtype that contains only the positive integers.
- constants. A constant is a data object that has a value that does not change, for example, "TRUE", "FALSE", or a physical constant such as "pi".
- functions. A function or predicate is declared for use.

2.4.1 Naming Rules

The rules for naming the type names, subtype names, constants, and functions are described in this section. The restrictions on the names allow the user to reference an item in the specification unit unambiguously.

- Type names must be uniquely identified and may not be used as a subtype name or a data object name.
- Subtype names, including the stimulus response components, must be uniquely identified by their name and may not be used as a type name or a data object name.
- Constants must be uniquely identified by their name and type. The name of a data object may be re-used if the type name is different for each data object using the same name.
- Functions and Predicates must be uniquely identified by the body, parameter placement, and return type.

3. DSPEC User Commands and Error Messages

This section describes the user command options available and the error messages displayed by DSPEC. The examples use the syntax of the tool support developed to demonstrate the commands.

3.1 User Commands

3.1.2 Introduction

There are three options that are supported by DSPEC. The first option allows the user to parse a requirements specification unit written in the DSPEC notation. The second option allows the user to apply all validation checks that apply to the data specification. The third option allows the user to translate a requirements specification unit written in DSPEC into the S notation.
3.1.3 Parse Requirements Specification

The user may determine if the requirements specification unit conforms to the DSPEC syntax. If the requirements specification conforms to the grammar, the tool does not display any messages. If the requirements specification does not conform to the grammar, an error message is reported to the user along with the line number the error is detected on. In the example below (Refer to Figure 3.1) the user is requesting to parse the specification in the file search_catalogue.v1. Error messages are documented in Section 3.2.

```
cascade<23>DSPEC search_catalogue.v1
cascade<24>
```

a. Successful Parsing Request

```
cascade<24>DSPEC search_catalogue.v1
ERROR: parse error detected near '1)'
    line: 6
```

b. Unsuccessful Parsing Request

Figure 3.1 a. Successful Parsing Request, b. Unsuccessful Parsing Request

3.1.4 Validate Requirements Specification

The user may determine if the requirements specification unit conforms to the DSPEC grammar and passes the validation checks by executing the DSPEC tool with the -v switch. In the example below (Figure 3.1), the user is requesting to parse the requirements specification in the file search_catalogue.v1 and determine if the requirements pass the validation checks.

If the requirements specification conforms to the grammar and passes the validation checks, the tool does not display any messages (Figure 3.2a). If the requirements specification does not conform to the grammar or does not pass a validation check, an error message is reported to the user along with the line number the error is detected (Figure 3.2b). In this example, the user has accidentally typed in the word “the” twice when using a constant on line 34. As a result, the tool cannot find a declaration that matches and reports the error. Error messages are documented in Section 3.4.

3.1.5 Translate Requirements Specification into S

The user may translate a requirements specification unit to S by executing the DSPEC
Appendix B. Definition of the Data Specification Notation

cascade<23>DSPEC -v search_catalogue.v1
cascade<24>

a. Successful Validation Request

cascade<24>DSPEC -v search_catalogue.v1
ERROR: declaration missing
   The the search response is sorted alphabetically
   line: 34
cascade<25>

b. Unsuccessful Validation Request

Figure 3.1 a. Successful Validation Request, b. Unsuccessful Validation Request

tool with the -t switch. In the example below (Figure 3.1), the user is requesting to translate the

cascade<23>DSPEC -t search_catalogue.v1
cascade<24>

a. Successful Translation Request

cascade<24>DSPEC -t search_catalogue.v1
ERROR: declaration missing
   The the search response is sorted alphabetically
   line: 34
cascade<25>

b. Unsuccessful Translation Request

Figure 3.1 a. Successful Translation Request, b. Unsuccessful Translation Request

requirements specification in the file search_catalogue.v1 into S.

If the requirements specification conforms to the grammar, passes the validation checks, and translates to S, the tool does not generate any messages (Figure 3.3a). If successful, the translation to S is written to an ASCII file called translate.s in the current working directory. If the file translate.s already exists, the old file is overwritten. The translate.s file passes
Appendix B. Definition of the Data Specification Notation

typechecking using the S typechecking tool FUSS [1], version 2.51.

If the requirements specification does not conform to the grammar, pass a validation check, or translate to S, an error message is reported to the user and the program exits (Figure 3.3b). If the validation checks are successful, the only way a translation to S can fail is due to insufficient memory. Error messages are documented in Section 3.4.

3.2 Error messages

A failure on parsing, performing validation checks, or translating the requirements to S is reported to the user with a brief textual message. A line number is also reported when the error is due to a problem in the requirements specification file. Line numbers are not reported when the error is due to insufficient memory.

Expected parsing, validation, and translation errors are described in Table 3:. The error messages output to the user all begin with the word ERROR and indicate a problem with the input requirements specification or the detection of a resource limitation (memory). In the table, the error messages are sorted alphabetically. Error messages that begin with the words INTERNAL ERROR are unexpected errors and indicate the tool has entered an unexpected state. These errors should be reported to the system maintainer.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration missing</td>
<td>A data type or object is used that has not been declared and is not built-in.</td>
</tr>
<tr>
<td>duplicate object</td>
<td>The object has already been declared. Duplicate objects are not allowed.</td>
</tr>
<tr>
<td>duplicate static association</td>
<td>The static association has already been declared. Duplicate static associations are not allowed.</td>
</tr>
<tr>
<td>duplicate subtype name</td>
<td>The subtype name has already been declared as a subtype name. Subtype names must be unique in the context.</td>
</tr>
<tr>
<td>duplicate type name</td>
<td>The type name has already been declared. Type names must be unique in the context.</td>
</tr>
<tr>
<td>file name missing</td>
<td>The command line interface requires a file name as an argument.</td>
</tr>
<tr>
<td>file not found</td>
<td>The input file name does not exist in the current working directory.</td>
</tr>
<tr>
<td>function already declared</td>
<td>The function has already been declared in the association or function section. Duplicate functions are not allowed.</td>
</tr>
</tbody>
</table>

Table 3: Error Messages in DSPEC
# Appendix B. Definition of the Data Specification Notation

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function or predicate declaration</td>
<td>A function or predicate has been applied but not declared and is not built-in.</td>
</tr>
<tr>
<td>missing</td>
<td></td>
</tr>
<tr>
<td>incorrect switch provided</td>
<td>The switch provided by the user is not supported by the tool. The two switches supported are -v (validate) and -t (translate to S).</td>
</tr>
<tr>
<td>input string too long, limit 1024</td>
<td>A comment, phrase, or system name has exceeded the maximum length.</td>
</tr>
<tr>
<td>insufficient memory</td>
<td>The program does not have enough memory to run. The heap has been exhausted.</td>
</tr>
<tr>
<td>logical condition not of type bool</td>
<td>A logical condition does not evaluate to type bool.</td>
</tr>
<tr>
<td>missing closing brace</td>
<td>A logical condition is missing a closing end brace.</td>
</tr>
<tr>
<td>multiple matches found</td>
<td>Multiple matches to a name have been found. The name is ambiguous.</td>
</tr>
<tr>
<td>multiple relationships found</td>
<td>Two or more ways of referencing the data have been found. The data reference is ambiguous.</td>
</tr>
<tr>
<td>no relationship found</td>
<td>Zero ways of referencing the data have been found by building paths. The data reference is ambiguous.</td>
</tr>
<tr>
<td>parameter must be angle bracket</td>
<td>A parameter in a function or predicate declaration is not delimited by angle brackets.</td>
</tr>
<tr>
<td>delimited</td>
<td></td>
</tr>
<tr>
<td>parse error detected near</td>
<td>A general error message that indicates the requirements specification does not conform to the grammar.</td>
</tr>
<tr>
<td>static association must relate types</td>
<td>The static association declared has a data object name. The association must be between data types.</td>
</tr>
<tr>
<td>static association not relating two</td>
<td>The static association declared has the type name associated with itself. The two types must be different.</td>
</tr>
<tr>
<td>unique types</td>
<td></td>
</tr>
<tr>
<td>subtype name already declared as a</td>
<td>The subtype name has already been declared as a type name. Subtype names must be unique in the context.</td>
</tr>
<tr>
<td>type name</td>
<td></td>
</tr>
<tr>
<td>type mismatch</td>
<td>The actual type does not match the expected type.</td>
</tr>
</tbody>
</table>

| Table 3: Error Messages in DSPEC    |                                                                                   |
4. Introduction to a Data Specification Unit in DSPEC

A data specification written in DSPEC introduces a context and supports the evaluation of data expressions and logical conditions based on that context. A specification is structured into seven sections (refer to Figure 4.1)

![Diagram of DSPEC Specification Structure](image)

The context is built using the data type, data subtype, data objects, associations, and function declarations. The data expressions and logical conditions are evaluated with respect to the context. Rules include type checking, type inference, and determining if the data references are unambiguous. The logical conditions are a special case of a data expression because they must evaluate to type bool.
5. Evaluation Rules

The set of rules used to determine if a data reference is unambiguous is described in this section using examples and pseudocode. There are two rules defined: 1) parent rule; 2) building paths rule. These rules are applied when a request is made that requires DSPEC to build all possible paths from one data type to another data type. The rules are used in the order listed above. For example, the first evaluation is done using the parent rule. If this is successful, then the evaluation for the data reference succeeds. If the parent rule evaluation is not successful, then the build paths rule is applied in the evaluation. If the evaluation using the building paths rule is successful, then the evaluation is successful. Otherwise, the evaluation fails.

A small, DSPEC context is used in this section to provide a working example. The context is composed of:

Data Types:
1) :<this type>.
2) :<that type>.
3) :<the other type>.
4) :<some type>.
5) :<some other type>.
6) :<integer>.

Data Objects:
1) important_constant:<integer>.
2) important_constant:<some other type>.
3) extremely_important_constant:<this type>.

Function Declarations:
1) "my function body"(<this type>) : <that type>.
2) "my other function body"(<this type>) : <that type>.
3) "another function body" (<the other type>) : <this type>.
4) "some function body" (<this type>) : <the other type>.
Appendix B. Definition of the Data Specification Notation

5) "a function body" (<the other type>) : <some type>.

6) "this function body" (<some type>) : <some other type>.

The directed graph of the context is derived from these declarations (refer to Figure 5.1). The nodes in the graph represent the data types. The arcs in the graph represent the functions.

![Directed Graph of Data Specification Example](image)

Figure 5.1 Directed Graph of Data Specification Example

### 5.1 Parent Rule

The parent rule is used to determine if there is exactly one parent child relationship in the directed graph representing the context. If the parent rule applies, the data reference is unambiguous. For example, the parent rule applied to the data references summarized in Table 4.
### Table 4: Example of Applying the Parent Rule

<table>
<thead>
<tr>
<th>From Data Reference</th>
<th>To Data Reference</th>
<th>Parent Rule Applies</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;that type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;some type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;some other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;this type&gt;</td>
<td>&lt;integer&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;this type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;some type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;some other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;that type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;integer&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;the other type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;some other type&gt;</td>
<td>Yes</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;this type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;that type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;the other type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;integer&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some type&gt;</td>
<td>&lt;some type&gt;</td>
<td>No</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>any type</td>
<td>No</td>
</tr>
<tr>
<td>&lt;some other type&gt;</td>
<td>any type</td>
<td>No</td>
</tr>
<tr>
<td>&lt;that type&gt;</td>
<td>any type</td>
<td>No</td>
</tr>
</tbody>
</table>

The pseudocode for the parent rule is:

get from type

get to type

initialize number of paths = 0
for every function declaration

if ((domain = from type) and (range = to type))

increment number of paths

return number of paths

If the number of paths returned is equal to one, then the parent rule applies and the data reference is unambiguous. If the number of paths returned is greater than one, then the parent rule does not apply and the data reference is ambiguous. If the number of paths is zero, then the parent rule does not apply. In this case, the data reference is not known to be ambiguous or unambiguous. Another rule, the building paths rule, needs to be applied to determine if there is a unique path in the graph.

5.2 Building Paths Rule

The building paths rule uses a depth first search to determine the paths in the graph from the from data type to the to data type. The search starts at the from type. Every possible path is attempted to reach the to data type in the data reference. For example, if the from data type is <the other type> and the to type is <some other type>, the following path is found in the graph (refer to Figure 5.1):

path 1

from <the other type> to <some type>
from <some type> to <some other type>

Since exactly one path is found, this data reference is unambiguous.

The graph is not restricted to being an acyclic graph and an additional check must be used in the search to determine if a path has a cycle in it. If this check is not made, then the data reference may have an infinite number of paths and the search does not terminate. If a cycle is detected, then the path currently being built is added to the paths and the next possible path is attempted. For example, if the from data type is <this type> and the to data type is <some other type>, then the following paths are found in the graph:

path 1
from < this type> to <the other type>
from <the other type> to <some type>
from <some type> to <some other type>

path 2
Appendix B. Definition of the Data Specification Notation

from *this type* to *the other type*

from *the other type* to *this type*

from *this type* to *the other type*

from *the other type* to *some type*

from *some type* to *some other type*

Since more than one path is found, this data reference from *this type* to *some other type* is ambiguous. In this example, the loop in the graph is detected when the second path is built.
6. Syntax and Validation Checks for a Data Specification Written in DSPEC

The seven sections for a data specification written in DSPEC are described in detail below. For each section, a synopsis, concrete syntax, and the validation checks that may be performed are provided.

6.1 General Remarks

All seven sections are local to the specification. The built-in types and objects (e.g., <bool>, TRUE) are available for use in every data specification.

The BNF conventions used in the definition of the language are listed below:
1. Reserved symbols are shown in bold-face type, e.g., the
2. Non-terminals are underlined, e.g., Logical Condition Section
3. Terminals, or tokens, are in plain text, e.g., phrase
4. A syntactic form enclosed within "{" and "}" indicates that the syntactic form is optional
5. A syntactic form enclosed within "{" and "}" followed by a "*" indicates zero or more iterations of the syntactic form
6. Required punctuation characters are enclosed within single quotes
7. A syntactic form enclosed within "{" and "}" followed by a "+" indicates one or more iterations of the syntactic form
8. The symbols "{", "}" used to specify syntactic forms should not be confused with the reserved symbols "{", "}". The latter are shown in bold-face in this document

The subsections that defined the language are not applicable at all times. The definition should be read with the following rules:
1. If a specification passes the lexical analysis (i.e., parses correctly), then the evaluation rules are applicable
2. If the specification passes the evaluation rules, then the specification may be translated into S

6.2 Data Specification

6.2.1 Synopsis

A data specification describes the context (type names, subtypes, data objects, associations, and function declarations) as well as the data expressions and logical conditions that are to be evaluated.

6.2.2 Concrete Syntax

A data specification unit must match the following BNF rules:

```plaintext
specification unit :=
    Type Names Section
    Subtype Names Section
    Data Objects Section
```
Appendix B. Definition of the Data Specification Notation

Associations Section
Function Section
Data Expression Section
Logical Condition Section

6.2.3 Validation Checking

The data specification fails validation checking if any of its sections fail.

6.3 Type Names Section

6.3.1 Synopsis

The type names section introduces zero or more data types for use. The data types may be a basic type or a set of a basic type.

6.3.2 Concrete Syntax

A type names section must match the following BNF rules:

\[
\text{Type Names Section} := \\
\text{Type Names:} \\
\{ \text{label ':' type name '.'} \\
| \text{label ':' set of type name '.'} \}^* \\
\text{type name} := \langle \text{phrase 'greater than'} \rangle \\
\text{set of type name} := \text{set of } \langle \text{phrase 'greater than'} \rangle \\
\]

6.3.3 Validation Checking

A type name declaration fails if:
1. the type_name is already declared.
2. the set_of_type_name is already declared.

6.4 Subtype Names Section

6.4.1 Synopsis

The subtype section introduces zero or more subtypes for use. The subtype names introduced may be subtypes of a basic type or subtypes of a set of basic types.

6.4.2 Concrete Syntax

A type names section must match the following BNF rules:

\[
\text{Subtype Section} := \\
\text{Subtypes:} \\
\]
Appendix B. Definition of the Data Specification Notation

{label A subtype name is a subtype of type name '.'
 | label A set subtype name is a subtype of set type name '.' } *

subtype name := '<' phrase '>'
type name := '<' phrase '>'
set subtype name := set of '<' phrase '>
set type name := set of '<' phrase '>

Type Names Section

6.4.3 Validation Checking

The subtype fails validation checking if:
1. the subtype_name is already declared for another type name
2. the subtype_name is already declared for the type name
3. the type_name is not declared
1. the set_subtype_name is already declared for another set type name
2. the set_subtype_name is already declared for the set_type_name
3. the set_type_name is not declared

6.5 Data Object Section

6.5.1 Synopsis

The data object section is composed of a list of zero or more constants. The constant declaration subsection introduces data objects of a specific type.

6.5.2 Concrete Syntax

A data object section must match the following BNF rules:

Data Object Section :=
Data Objects ':'
{label data object name ':' type name ' .' } *

data object name := phrase | "" phrase "" | '<' phrase '>
type name := '<' phrase '>'

6.5.3 Validation Checking

The data object declaration fails if:
1. the type_name is not declared
2. the data_object_name is already declared for the type_name
6.6 Association Section

6.6.1 Synopsis

The association section introduces relationships between types of data objects. The “one or more” and “zero or more” cardinalities describe an association from the type\_name\_1 or set\_type\_name\_1 to a set of type\_name\_2. The “exactly one” and “zero or one” cardinalities describe an association from the type\_name\_1 or set\_type\_name\_1 to the type\_name\_2. The static associations are templated, local function or predicate declarations.

6.6.2 Concrete Syntax

An association section must match the following BNF rules:

association subsection :=

Associations:
{ label Every type\_name\_1 is associated with : { label association ‘;’ }+ label association ‘.’ } |
label Every set type\_name\_1 is associated with : { label association ‘;’ }+ label association ‘.’ |
label Every type\_name\_1 is associated with association ‘.’ }* |
label Every set type\_name\_1 is associated with association ‘.’ }* 

association := zero or one type\_name\_2 |
zero or more type\_name\_2 |
one or more type\_name\_2 |
exactly one type\_name\_2 

type\_name\_1 := ‘<’ phrase ‘>’
type\_name\_2 := ‘<’ phrase ‘>’

6.6.3 Validation Checking

The association declaration fails validation checking if:
1. a type\_name\_1 is not declared.
2. a type\_name\_2 is not declared.
3. a set type\_name\_1 is not declared.
4. a set type\_name\_2 is not declared.
5. an association is already declared for type\_name\_1 or its subtypes and type\_name\_2 or its subtypes.
6. an association is already declared for set type\_name\_1 or its subtypes and type\_name\_2 or its subtypes.
7. type\_name\_1 is declared in an “exactly one” or a “one or more” association with a type\_name\_2
that has the same as type name\textsubscript{1}.
8. set type name\textsubscript{1} is declared in a “one or more” or a “zero or more” association with a type name\textsubscript{2} that has the same base type as set type name\textsubscript{1}.

6.7 Function Section

6.7.1 Synopsis

The functions section introduces a list of functions and/or predicates for use. The functions are declared in flex-fix format.

6.7.2 Concrete Syntax

A function section must match the following BNF rules:

\[
\text{Function Section} := \\
\text{Functions:} \\
\{ \text{label function is a function ':' return type name '.'} \}^* \\
\text{function} := \\
\{ \{ \text{parameter } \}^+ \text{ function body text } \{ \text{parameter } \}^* \}^+ \\
\| \{ \text{function body text } \{ \text{parameter } \}^+ \}^+ \{ \text{function body text } \} \\
\text{return type name} := ' '<\text{ phrase } '>' | \text{set of ' '<\text{ phrase } '>' } \\
\text{parameter} := '( '<\text{ phrase } '>' )' | '(' \text{set of ' '<\text{ phrase } '>' } ')' \\
\text{function body text} := \{ \text{phrase } \}^+ \\
\]

6.7.3 Validation Checking

The function section fails typechecking if:
1. the return_type Name is not declared.
2. the function is already declared.
3. a parameter is not declared as a data type

6.8 Data Expression Section

6.8.1 Synopsis

A data expression denotes a data object or a set of data objects.

6.8.2 Concrete Syntax

A data expression section must match the following BNF rules:

\[
\text{Data Expression Section} := \\
| \text{integer} \\
| \text{the '{' message '}'} \text{ associated with '{' data expression '}'} \\
\]

252
Appendix B. Definition of the Data Specification Notation

| '{ { the } message }' |
| function_application |

function_application :=

{{ parameter }+ function body text { parameter }* }+ 

| {{ function body text. } { parameter }+ }+ { function body text } |

| {{ function body text } { parameter }+ }+ |

parameter := '{' data expression '}'

function body text := { phrase }+

title thread := phrase

message := phrase | '<' phrase '>' | "" phrase "" |

set of '<' phrase '>'

6.8.3 Validation Checking

The data expression fails if:
1. A message is not a declared data type, subtype, constant, variable system parameter, system state component, or integer
2. A function declaration does not match the function application
3. A message is not uniquely related to the data expression, if present
4. A message is not uniquely related to the input, if the data expression is not present
5. The evaluation of the data references are unambiguous

6.9 Logical Condition Section

6.9.1 Synopsis

A logical condition is an expression that denotes a condition that is either true or false.

6.9.2 Concrete Syntax

A logical condition section must match the following BNF rules:

Logical Condition Section :=

all of the following conditions are satisfied : Labelled condition list

| one or more of the following conditions are satisfied : Labelled condition list |

| not '{ logical condition ' |

| '{ logical condition ' and '{ logical condition ' |

| '{ logical condition ' or '{ logical condition ' |

| '{ predicate application' |

| table application |

| '{ message ' |

// quantified conditions

| one or more of the '{ message ' such that '{ logical_condition ' |

| all of the '{ message ' such that '{ logical_condition ' |

| none of the '{ message ' such that '{ logical_condition ' |

253
Appendix B. Definition of the Data Specification Notation

message := phrase | '<' phrase '>' | "" phrase "" | set of '<' phrase '>'

Labelled condition list :=
   { label '{' logical condition '}' ';' }+
   label '{' logical condition '}'

predicate application :=
   {{ parameter }+ predicate body text { parameter }* }+
   | {{ predicate body text } { parameter }+ }+ { predicate body text }
   | {{ predicate body text } { parameter }+ }+ as specified in the 'V title thread 'V

table application :=
   { { parameter }+ { predicate ';' }* }+
   table name

table name := phrase

parameter := '{' data expression '}'
function body text := { phrase }+
title thread := phrase
message := phrase | '<' phrase '>' | "" phrase ""
table name := phrase

6.9.3 Validation Checking

The logical condition fails if:
1. A message is not declared of type bool, a subtype of base type bool, constant of type bool or a
   subtype with base type bool, variable system parameter of type bool or a subtype of base type
   bool, or system state component of type bool or a subtype of base type bool
2. a predicate declaration does not match a predicate application
3. a table_declaration does not match a table_application
4. the evaluation of a parameter in a predicate_application or a table_application is unambiguous
5. in a quantified expression, the message is not a type or subtype declaration.
Appendix C. Experimental Design and Results

This Appendix provides the detailed experimental design used to evaluate the SRRS notation and the results.

1.1 Experimental Design

1.1.1 Introduction

A wide variety of software requirement specification approaches are available for use in industry that range from informal natural language to formal notations such as Z. For large scale systems, structured natural language approaches are used because they require minimal training, and the requirements are straightforward to write and review. The limitation of a structured natural approach is that there is very limited tool support available. The natural language documents are not amenable to unambiguous parsing, typechecking, or other analysis using tool support.

To obtain the parsing, typechecking and semantic analysis support a formal notation must be selected. Introducing a formal method is met with resistance because they are perceived to be expensive to adopt: notations such as Z are not considered readable and engineers require training in the underlying mathematics, the notation, and the tool support available. The uncertainty involved in the adoption of a formal notation is not outweighed by the potential benefits of applying the method as there is very little experimental data available.

The purpose of this experiment is to compare a formal version of a software requirements specification notation to its semi-formal version. The two notations draw from the Threads-Capabilities method, a variation of a use case technique, used at Hughes Aircraft of Canada Ltd. [HCS93]. The two notations are compared in terms of the quality of the specifications written and the effort required for training, writing, reviewing, and correcting specification units.

1.1.2 Problem Statement

The costs and benefits of introducing a new, formal requirements specification approach need to be shown experimentally before the approach is accepted for use in industry.

The costs of introducing a formal notation include additional formal and on the job training. To be useful, formal notations are supported by automated tools that may provide scanning, parsing, typechecking, and analysis to assist the author. This tool support assists the authors of specifications to detect and correct defects earlier in the development lifecycle. This reduces the cost of development because errors cost an order of magnitude more to detect and correct in each subsequent phase.

The formal Stimulus Response Requirement Specification notation is compared to the semi-formal Stimulus Response Requirements notation with respect to the training time, effort required to develop a set of specification units, and the quality of those specification units.
1.1.3 Objectives

The objectives of this experiment are to test the following hypotheses:

1. Ho: The use of a completely defined syntax and an automated parsing/typechecking/semantic analysis tool results in a requirement specification that has the same average detected defect rate per allocated requirement object than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

Ha: The use of a completely defined syntax and an automated parsing/typechecking/semantic analysis tool results in a requirement specification that has a lower average detected defect rate per allocated requirement object than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

2. Ho: The use of a completely defined syntax and an automated parsing/typechecking/semantic analysis tool results in a requirement specification that has the same average effort per allocated requirement object to describe than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

Ha: The use of a completely defined syntax and an automated parsing/typechecking/analysis tool results in a requirement specification that has a lower average effort per allocated requirement object to describe than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

3. Ho: The use of a completely defined syntax and an automated parsing/typechecking/semantic analysis tool results in the same training time per subject than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

Ha: The use of a completely defined syntax and an automated parsing/typechecking/analysis tool results in a greater average training time per subject than a requirement specification that does not use a completely defined syntax and an automated parsing/typechecking/semantic analysis tool.

The average detected defect rate is defined in terms of the total number of detected defects in the specification units divided by the total number of allocated requirement objects to the specification units written.

The average effort per allocated ROID is the total amount of time used to write, review, and correct the specification units written divided by the number of ROIDs allocated to the specification units written.

The average training effort is the amount of time used to train the subjects in the notation both formally (in the classroom) and informally (independent study) divided by the number of subjects.
1.1.4 Variables

The independent variables identified are:
1. The requirements specification notation used to write specification units
2. The number of specification units written
3. The number of ROIDs allocated to the specification units written
4. The number of subjects
5. The number of specification units written per subject

The dependent variables identified are:
1. The average defect detection rate per allocated ROID
2. The average effort to write, review, and correct per allocated ROID
3. training time

The treatment variable is the requirements specification notation used to write the specification units.

1.1.5 Experimental Design Constraints

The experimental design constraints identified are:
1. The budget for the experiment is fixed
2. The number of specification units that are written is 18
3. The number of iterations of review/defect correction is a maximum of 2
4. Each subject may only participate in one group for the experiment
5. The 18 specification units written are randomly selected from 52 specification units
6. The subjects are recruited for a paid position and are selected based on background and availability

1.1.6 Materials

The materials used in this experiment include training material, a defect checklist, data recording forms, the SLS for the system used in the experiment, a list of the identified specification unit titles, and the allocation of ROIDs from the SLS to the identified titles.

For the control and experimental group the training material includes a set of lecture slides, a SLS for a video rental system, a list of identified specification units for the video rental system, and the allocation of the ROIDs to the identified titles. The video system material is used in a hands-on exercise in the training session. The experimental group also receives a user manual and a tutorial for the SRRS tool. The training materials are in [Coo99].

The lecture slides are organized into four modules as follows:

Module 1: Overview of the Software Development Lifecycle
Software Development Lifecycles (SDLCs)
Appendix C. Experimental Design and Results

Introduction to Requirements

Module 2: Initial Phases of Software Development Lifecycle
System Level Specification (SLS)
Software Requirement Specification (SRS)
Requirements Traceability & Management

Module 3: Describing the Functional Requirements
Stimulus Response Requirements Specification (SRRS) Approach
Developing the Functional Requirements

Module 4: The Requirements to System Test Interface
System Integration Testing
Test Case Generation

There is a checklist used in the experiment to assist the reviewers in detecting and categorizing defects. A sample of the defect checklist form is provided below:
## Defect Checklist

<table>
<thead>
<tr>
<th>Category</th>
<th>Description of Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Errors</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>standardized phrase is incorrect/missing</td>
</tr>
<tr>
<td>S2</td>
<td>section/sub-section title is incorrect/missing</td>
</tr>
<tr>
<td>S3</td>
<td>response verb is not from constrained list</td>
</tr>
<tr>
<td>S4</td>
<td>minor grammatical errors including punctuation, misuse of keyword, subsection title incorrect.</td>
</tr>
<tr>
<td>S5</td>
<td>list structure incorrect/missing</td>
</tr>
<tr>
<td>S6</td>
<td>did not use list when required</td>
</tr>
<tr>
<td>S7</td>
<td>other</td>
</tr>
<tr>
<td>Type Violations:</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>type name declaration missing</td>
</tr>
<tr>
<td>T2</td>
<td>type alias declaration missing</td>
</tr>
<tr>
<td>T3</td>
<td>constant declaration missing</td>
</tr>
<tr>
<td>T4</td>
<td>variable system parameter declaration missing</td>
</tr>
<tr>
<td>T5</td>
<td>system state component declaration missing</td>
</tr>
<tr>
<td>T6</td>
<td>stimulus used but not introduced</td>
</tr>
<tr>
<td>T7</td>
<td>stimulus used but not introduced</td>
</tr>
<tr>
<td>T8</td>
<td>other</td>
</tr>
<tr>
<td>Analysis Errors</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>type name declared but not used</td>
</tr>
<tr>
<td>A2</td>
<td>type alias name declared but not used</td>
</tr>
<tr>
<td>A3</td>
<td>constant declared but not used</td>
</tr>
<tr>
<td>A4</td>
<td>variable system parameter declared but not used</td>
</tr>
<tr>
<td>A5</td>
<td>system state component declared but not used</td>
</tr>
<tr>
<td>A6</td>
<td>stimulus introduced but not used</td>
</tr>
<tr>
<td>A7</td>
<td>response introduced but not used</td>
</tr>
<tr>
<td>A8</td>
<td>committed data types not declared as system state components</td>
</tr>
<tr>
<td>A9</td>
<td>static association missing</td>
</tr>
<tr>
<td>A10</td>
<td>static association introduced but not used</td>
</tr>
<tr>
<td>A11</td>
<td>incorrect stimuli used in requirement</td>
</tr>
<tr>
<td>A12</td>
<td>incorrect response used in requirement</td>
</tr>
<tr>
<td>A13</td>
<td>incorrect data used in requirement</td>
</tr>
<tr>
<td>A14</td>
<td>design detail</td>
</tr>
<tr>
<td>A15</td>
<td>requirement allocation error</td>
</tr>
<tr>
<td>A16</td>
<td>requirement tracing error</td>
</tr>
<tr>
<td>A17</td>
<td>other</td>
</tr>
</tbody>
</table>

259
Two forms are used in the experiment to assist in the collection of data. The first form is the time recording form. A sample of this form is provided below:

<table>
<thead>
<tr>
<th>Date</th>
<th>Recorded by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C. Experimental Design and Results

The second data collection form is the defect recording. A sample of this form is provided below:

**Defect Recording Sheet**

<table>
<thead>
<tr>
<th>Date</th>
<th>Recorded by</th>
<th>Specification Unit Identifier</th>
<th>Specification Unit Version #</th>
<th>Writing specification or test case generation phase</th>
<th>Defect Recording sheet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Defect Identifier</th>
<th>Type of Error</th>
<th>Category of Error</th>
<th>Checklist Identifier</th>
<th>Text Description of Defect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The materials also include a SLS for the “Integrated On-line Library System” that is used in the experiment. The SLS is derived from a request for proposal for an integrated on-line library system [Cor87]. The SLS is composed of 652 ROIDs. Using the SLS, 52 specification units are identified and 432 ROIDs are allocated to them.
1.1.7 Methods

Selection of Subjects

The subjects for the control and the experimental groups are \textit{not} randomly selected from the applicants for the paid positions. Instead, the subjects are interviewed and selected based on their background (a specific course in software engineering), communication skills, and availability.

Selection, assignment, and order for writing specification units

The 18 specification units that are written are randomly selected from the outline of the Integrated On-line Library System software requirements specification document that is composed of 52 specification units.

The 18 specification units are randomly assigned to three groups. The groups of specification units are randomly assigned to the three subjects. Each subject writes six specification units.

The specification units are presented to a subject for writing one at a time.

The specification units are presented in the same order for both the control and experimental groups.

The specification units are reviewed by the author's peers.

Data Collection

The time is recorded every half hour by the subjects. The half hour of time may be broken down into 5 minute "chunks". These time sheets are \textit{not} to be used for payroll purposes.

The defect recording is done in the peer review of a specification unit. The author does not formally review their own work. The type of error, category of error, the checklist identifier, and a brief description of the defect are recorded on the defect recording list. The type of error is either an error of omission (recorded as "O") or an error of commission (recorded as a "C"). The category of the error can be any of the following: ambiguous information (recorded as "A"); extraneous information (recorded as "E"); inconsistent information (recorded as "II"); incorrect fact (recorded as "IF"); missing information (recorded as "MI"); redundant information ("RI"); or a miscellaneous defect (recorded as "MD"). The checklist identifier is the letter, number identifier from the defect checklist that matches the problem the best. The brief textual description of the defect is recorded to assist the author in understanding the defect detected.

Training

In this experiment, the formal training for both the control and the experimental groups is structured as a set of four modules that are presented in a lecture format. Module one provides a brief introduction to the Formalware project, reviews the waterfall and spiral software develop-
ment lifecycle models, and describes the characteristics of a “good” requirement. Module two covers the initial phases of the software development lifecycle. It clarifies the distinction between a system level specification (SLS) and a software requirements specification (SRS), describes the process to develop a SRS, and reviews traceability. Module three introduces the requirements specification notation used by the group and describes the six step process used to describe the functional requirements in an SRS (refer to the following section on the experimental procedure). Module four reviews the requirements to system level testing interface and emphasizes the impact the quality of an SRS has on the development of system level test cases.

Modules one, two, and four are exactly the same for the control and the experimental groups, however, module three is tailored. The third module in the training material for the group introduces the semi-formal stimulus response requirements specification notation to describe the functional requirements in an SRS. The training material for the experimental group introduces the formal stimulus response requirements specification notation. The six step process used to develop the specification units differs only in the use of the SRRS tool support by the experimental group in steps three and five.

**Experimental Procedure**

The six step process is a top down refinement process for writing a specification unit (refer to Figure 1.1). The inputs for the first step in the process are the SLS, the title of the specification unit to be developed, and a list of ROIDs that are allocated to the title. The first step in the process is to ensure that each of the allocated ROIDs should be allocated and to determine if any ROIDs are missing from the allocation. After the ROIDs are checked, the author drafts an outline of the specification unit in step two using an ASCII editor. The section headers are entered and each section is outlined in point form. In the third step, the author completes the version of the specification unit by refining the outline and filling in the details of the specification unit. The tool support in this step is also an ASCII editor. The experimental group also uses the SRRS tool support to validate their specification. In addition to completing the first version, a traceability report is generated to indicate how the allocated ROIDs have been addressed in the specification unit. The specification unit and the traceability report are submitted as a hardcopy for a peer review in step four. The author’s peers review the specification unit and record defects according to type, category, defect checklist identifier, and a brief description. When the peer review is complete the specification unit, traceability report, and the defect recording sheet(s) are returned to the author. The author corrects the specification unit and revises the traceability report as necessary in step five. The review cycle is complete in the experiment when the specification unit has no defects detected or when the review cycle has occurred twice. The final version is submitted in the last step of the process.

**Calculations**

The percent differences are calculated using Equation 1.

\[
\text{% difference} = \left( \frac{\text{control value} - \text{experimental value}}{\text{control value}} \right) \times 100
\]  

(EQ 1)
Appendix C. Experimental Design and Results

Figure 1.1 A Six Step Process to Develop a Specification Unit
1.2 Experimental Results

1.2.1 Data Collected

The experimental data is summarized in a set of six tables. Table 1 and Table 2 contain the effort to write, review, and correct the specification units for the control group and the experimental group respectively. Tables 3 and 4 contain the training time for the control group and the experimental group. Tables 5 and 6 contain the defects detected for the control and the experimental groups.
### Table 1: Group1 Time (write, review, correct)

<table>
<thead>
<tr>
<th>Specification Unit</th>
<th>write version 1 minutes</th>
<th>review version 1 minutes</th>
<th>correct version 1 minutes</th>
<th>review version 2 minutes</th>
<th>correct version 2 minutes</th>
<th># allocated ROIDs</th>
<th>time to write minutes</th>
<th>review cycle minutes</th>
<th>Total minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charge Item (interlibrary)</td>
<td>495</td>
<td>115</td>
<td>205</td>
<td>65</td>
<td>25</td>
<td>26</td>
<td>495</td>
<td>410</td>
<td>905</td>
</tr>
<tr>
<td>Create Booking on Item</td>
<td>390</td>
<td>110</td>
<td>40</td>
<td>45</td>
<td>20</td>
<td>18</td>
<td>390</td>
<td>215</td>
<td>605</td>
</tr>
<tr>
<td>Modify Acquisition Plan</td>
<td>415</td>
<td>225</td>
<td>45</td>
<td>45</td>
<td>25</td>
<td>18</td>
<td>415</td>
<td>340</td>
<td>755</td>
</tr>
<tr>
<td>Discharge Item (non-interlibrary)</td>
<td>340</td>
<td>155</td>
<td>30</td>
<td>45</td>
<td>15</td>
<td>16</td>
<td>340</td>
<td>245</td>
<td>585</td>
</tr>
<tr>
<td>Renew Item</td>
<td>450</td>
<td>80</td>
<td>50</td>
<td>65</td>
<td>10</td>
<td>21</td>
<td>450</td>
<td>205</td>
<td>655</td>
</tr>
<tr>
<td>View Bookings</td>
<td>210</td>
<td>115</td>
<td>30</td>
<td>35</td>
<td>10</td>
<td>43</td>
<td>210</td>
<td>190</td>
<td>400</td>
</tr>
<tr>
<td>Modify Borrower's Information</td>
<td>320</td>
<td>75</td>
<td>10</td>
<td>40</td>
<td>10</td>
<td>7</td>
<td>320</td>
<td>135</td>
<td>455</td>
</tr>
<tr>
<td>Search for Catalogue Item</td>
<td>185</td>
<td>95</td>
<td>15</td>
<td>85</td>
<td>35</td>
<td>27</td>
<td>185</td>
<td>230</td>
<td>415</td>
</tr>
<tr>
<td>Receive Acquisition</td>
<td>480</td>
<td>200</td>
<td>105</td>
<td>115</td>
<td>30</td>
<td>30</td>
<td>480</td>
<td>450</td>
<td>930</td>
</tr>
<tr>
<td>Modify Jobber/Publisher Matrix</td>
<td>125</td>
<td>170</td>
<td>25</td>
<td>80</td>
<td>15</td>
<td>3</td>
<td>125</td>
<td>290</td>
<td>415</td>
</tr>
<tr>
<td>Pay-for-use Database Access Requests</td>
<td>135</td>
<td>30</td>
<td>5</td>
<td>25</td>
<td>5</td>
<td>5</td>
<td>135</td>
<td>65</td>
<td>200</td>
</tr>
<tr>
<td>Request Statistics on Interlibrary Loan</td>
<td>100</td>
<td>60</td>
<td>30</td>
<td>45</td>
<td>35</td>
<td>4</td>
<td>100</td>
<td>170</td>
<td>270</td>
</tr>
<tr>
<td>Delete Catalogue Item</td>
<td>230</td>
<td>75</td>
<td>35</td>
<td>60</td>
<td>5</td>
<td>1</td>
<td>230</td>
<td>175</td>
<td>405</td>
</tr>
<tr>
<td>Add Catalogue Item</td>
<td>115</td>
<td>50</td>
<td>40</td>
<td>25</td>
<td>5</td>
<td>11</td>
<td>115</td>
<td>120</td>
<td>235</td>
</tr>
<tr>
<td>Request General Statistics</td>
<td>40</td>
<td>35</td>
<td>10</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>40</td>
<td>70</td>
<td>110</td>
</tr>
<tr>
<td>Receive Serial</td>
<td>425</td>
<td>145</td>
<td>50</td>
<td>70</td>
<td>15</td>
<td>7</td>
<td>425</td>
<td>280</td>
<td>705</td>
</tr>
<tr>
<td>Create Acquisition Plan</td>
<td>90</td>
<td>170</td>
<td>35</td>
<td>60</td>
<td>10</td>
<td>17</td>
<td>90</td>
<td>275</td>
<td>365</td>
</tr>
<tr>
<td>Modify On-line Adaptation Data</td>
<td>185</td>
<td>125</td>
<td>45</td>
<td>75</td>
<td>0</td>
<td>11</td>
<td>185</td>
<td>245</td>
<td>430</td>
</tr>
</tbody>
</table>

269  4730  4110  8840
<table>
<thead>
<tr>
<th>Specification Unit</th>
<th>write version 1 minutes</th>
<th>review version 1 minutes</th>
<th>correct version 1 minutes</th>
<th>review version 2 minutes</th>
<th>correct version 2 minutes</th>
<th># allocated ROIDs</th>
<th>time to write minutes</th>
<th>review cycle minutes</th>
<th>Total minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charge Item (interlibrary)</td>
<td>390</td>
<td>85</td>
<td>60</td>
<td>45</td>
<td>25</td>
<td>26</td>
<td>390</td>
<td>215</td>
<td>605</td>
</tr>
<tr>
<td>Create Booking on Item</td>
<td>275</td>
<td>140</td>
<td>120</td>
<td>75</td>
<td>60</td>
<td>18</td>
<td>275</td>
<td>395</td>
<td>670</td>
</tr>
<tr>
<td>Modify Acquisition Plan</td>
<td>250</td>
<td>170</td>
<td>50</td>
<td>65</td>
<td>55</td>
<td>18</td>
<td>250</td>
<td>340</td>
<td>590</td>
</tr>
<tr>
<td>Discharge Item (non-interlibrary)</td>
<td>195</td>
<td>120</td>
<td>45</td>
<td>50</td>
<td>0</td>
<td>16</td>
<td>195</td>
<td>215</td>
<td>410</td>
</tr>
<tr>
<td>Renew Item</td>
<td>220</td>
<td>90</td>
<td>30</td>
<td>90</td>
<td>35</td>
<td>21</td>
<td>220</td>
<td>245</td>
<td>465</td>
</tr>
<tr>
<td>View Bookings</td>
<td>415</td>
<td>30</td>
<td>70</td>
<td>40</td>
<td>25</td>
<td>43</td>
<td>415</td>
<td>165</td>
<td>580</td>
</tr>
<tr>
<td>Modify Borrower’s Information</td>
<td>150</td>
<td>75</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>7</td>
<td>150</td>
<td>120</td>
<td>270</td>
</tr>
<tr>
<td>Search for Catalogue Item</td>
<td>125</td>
<td>45</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>27</td>
<td>125</td>
<td>75</td>
<td>200</td>
</tr>
<tr>
<td>Receive Acquisition</td>
<td>90</td>
<td>90</td>
<td>30</td>
<td>40</td>
<td>0</td>
<td>30</td>
<td>90</td>
<td>160</td>
<td>250</td>
</tr>
<tr>
<td>Modify Jobber/Publisher Matrix</td>
<td>65</td>
<td>50</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>3</td>
<td>65</td>
<td>65</td>
<td>130</td>
</tr>
<tr>
<td>Pay-for-use Database Access Requests</td>
<td>115</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>115</td>
<td>35</td>
<td>150</td>
</tr>
<tr>
<td>Request Statistics on Interlibrary Loan</td>
<td>30</td>
<td>30</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>30</td>
<td>45</td>
<td>75</td>
</tr>
<tr>
<td>Delete Catalogue Item</td>
<td>30</td>
<td>45</td>
<td>15</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>30</td>
<td>65</td>
<td>95</td>
</tr>
<tr>
<td>Add Catalogue Item</td>
<td>50</td>
<td>60</td>
<td>25</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>50</td>
<td>105</td>
<td>155</td>
</tr>
<tr>
<td>Request General Statistics</td>
<td>80</td>
<td>20</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>4</td>
<td>80</td>
<td>50</td>
<td>130</td>
</tr>
<tr>
<td>Receive Serial</td>
<td>180</td>
<td>60</td>
<td>15</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td>180</td>
<td>100</td>
<td>280</td>
</tr>
<tr>
<td>Create Acquisition Plan</td>
<td>120</td>
<td>30</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>17</td>
<td>120</td>
<td>60</td>
<td>180</td>
</tr>
<tr>
<td>Modify On-line Adaptation Data</td>
<td>65</td>
<td>45</td>
<td>15</td>
<td>25</td>
<td>5</td>
<td>11</td>
<td>65</td>
<td>80</td>
<td>145</td>
</tr>
</tbody>
</table>

|                                |                         |                          |                          |                          |                          |                  |                      |                     |              |
|                                | 269                     | 2845                     | 2535                     | 5380                     |

Table 2: Group 2 Time (write, review, correct)
### Appendix C. Experimental Design and Results

#### Table 3: Group 1 Training Time

<table>
<thead>
<tr>
<th>Group 1</th>
<th>Formal Training Time minutes</th>
<th>Informal Training Time minutes</th>
<th>Total Training Time minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject 1</td>
<td>420</td>
<td>25</td>
<td>445</td>
</tr>
<tr>
<td>subject 2</td>
<td>420</td>
<td>40</td>
<td>460</td>
</tr>
<tr>
<td>subject 3</td>
<td>420</td>
<td>20</td>
<td>440</td>
</tr>
<tr>
<td>Totals</td>
<td>1260</td>
<td>85</td>
<td>1345</td>
</tr>
</tbody>
</table>

#### Table 4: Group 2 Training Time

<table>
<thead>
<tr>
<th>Group 2</th>
<th>Formal Training Time minutes</th>
<th>Informal Training Time minutes</th>
<th>Total Training Time minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject 1</td>
<td>835</td>
<td>255</td>
<td>1090</td>
</tr>
<tr>
<td>subject 2</td>
<td>835</td>
<td>220</td>
<td>1055</td>
</tr>
<tr>
<td>subject 3</td>
<td>835</td>
<td>875</td>
<td>1710</td>
</tr>
<tr>
<td>Totals</td>
<td>2505</td>
<td>1350</td>
<td>3855</td>
</tr>
</tbody>
</table>
### Table 5: Group 1 Defects

<table>
<thead>
<tr>
<th>Specification Unit</th>
<th># Syntax Defects Version 1</th>
<th># Type Defects Version 1</th>
<th># Analysis Defects Version 1</th>
<th># Syntax Defects Version 2</th>
<th># Type Defects Version 2</th>
<th># Analysis Defects Version 2</th>
<th># Total Syntax Defects</th>
<th># Total Type Defects</th>
<th># Total Analysis Defects</th>
<th>Total Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charge Item (interlibrary)</td>
<td>23</td>
<td>23</td>
<td>15</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>32</td>
<td>29</td>
<td>20</td>
<td>81</td>
</tr>
<tr>
<td>Create Booking on Item</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>Modify Acquisition Plan</td>
<td>11</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>1</td>
<td>5</td>
<td>31</td>
<td>24</td>
<td>23</td>
<td>78</td>
</tr>
<tr>
<td>Discharge Item</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>Renew Item</td>
<td>17</td>
<td>6</td>
<td>13</td>
<td>9</td>
<td>0</td>
<td>10</td>
<td>26</td>
<td>6</td>
<td>23</td>
<td>55</td>
</tr>
<tr>
<td>View Bookings</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>13</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>Modify Borrower’s Information</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>Search for Catalogue Item</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>Receive Acquisition</td>
<td>25</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>9</td>
<td>8</td>
<td>53</td>
<td>29</td>
<td>32</td>
<td>114</td>
</tr>
<tr>
<td>Modify Jobber/Publisher Matrix</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>18</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>Pay-for-use database Requests</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Request Statistics on Interlibrary Loan</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>15</td>
<td>8</td>
<td>16</td>
<td>39</td>
</tr>
<tr>
<td>Delete Catalogue Item</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>Add Catalogue Item</td>
<td>8</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>13</td>
<td>5</td>
<td>15</td>
<td>33</td>
</tr>
<tr>
<td>Request General Statistics</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Receive Serial</td>
<td>6</td>
<td>7</td>
<td>15</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>22</td>
<td>42</td>
</tr>
<tr>
<td>Create Acquisition Plan</td>
<td>11</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>16</td>
<td>7</td>
<td>9</td>
<td>32</td>
</tr>
<tr>
<td>Modify on-line adaptation data</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>19</td>
<td>11</td>
<td>15</td>
<td>45</td>
</tr>
</tbody>
</table>

|                             | 267                          | 199                       | 237                           | 703                         |
## Appendix C. Experimental Design and Results

<table>
<thead>
<tr>
<th>Specification Unit:</th>
<th>Syntax Defects</th>
<th>Analysis Defects</th>
<th>Total Type Defects</th>
<th>Syntax Defects</th>
<th>Analysis Defects</th>
<th>Total Type Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charge Item (interlibrary)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Create Booking on Item</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Modify Acquisition Plan</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Discharge Item</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Renew Item</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>View Bookings</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Modify Borrower's Information</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Search for Catalogue Item</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Receive Acquisition</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Modify Jollier/Publisher Matrix</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pay-for-use database Requests</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Request Statistics on Interlibrary Loan</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Delete Catalogue Item</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add Catalogue Item</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Request General Statistics</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Receive Serial</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Create Acquisition Plan</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Modify on-line adaptation data</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6: Group 2 Defects
Appendix C. Experimental Design and Results

<table>
<thead>
<tr>
<th>Specification Unit Identifier (title)</th>
<th>Subject Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charge Item (Interlibrary)</td>
<td>1</td>
</tr>
<tr>
<td>Create Booking on Item</td>
<td>2</td>
</tr>
<tr>
<td>Modify Acquisition Plan</td>
<td>3</td>
</tr>
<tr>
<td>Discharge Item (non-interlibrary)</td>
<td>2</td>
</tr>
<tr>
<td>Renew Item</td>
<td>3</td>
</tr>
<tr>
<td>View Bookings</td>
<td>1</td>
</tr>
<tr>
<td>Modify Borrower’s Information</td>
<td>2</td>
</tr>
<tr>
<td>Search for Catalogue Item</td>
<td>1</td>
</tr>
<tr>
<td>Receive Acquisition</td>
<td>3</td>
</tr>
<tr>
<td>Modify Jobber/Publisher Matrix</td>
<td>1</td>
</tr>
<tr>
<td>Pay-for-use Database Access Requests</td>
<td>2</td>
</tr>
<tr>
<td>Request Statistics on Interlibrary Loan</td>
<td>3</td>
</tr>
<tr>
<td>Delete Catalogue Item</td>
<td>2</td>
</tr>
<tr>
<td>Add Catalogue Item</td>
<td>3</td>
</tr>
<tr>
<td>Request General Statistics</td>
<td>1</td>
</tr>
<tr>
<td>Receive Serial</td>
<td>2</td>
</tr>
<tr>
<td>Create Acquisition Plan</td>
<td>1</td>
</tr>
<tr>
<td>Modify on-line Adaptation Data</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7: Random Assignment of Specification Units, Subjects

1.2.2 Analysis of Data

Percentage Difference Calculations

The experimental results for the defect rates are summarized in Table 8. The results show a reduction in the syntax, type, and the analysis defects detected for Group 2. The % difference between the group using the semi-formal notation (Group 1) and the group using the formal notation (Group 2) for the total number of defects detected per allocated ROID shows an 81%
reduction in detected defects.

<table>
<thead>
<tr>
<th></th>
<th>Number of syntax defects per ROID</th>
<th>Number of type defects per ROID</th>
<th>Number of analysis defects per ROID</th>
<th>Number of total defects per ROID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>0.99</td>
<td>0.74</td>
<td>0.88</td>
<td>2.61</td>
</tr>
<tr>
<td>Group 2</td>
<td>0.09</td>
<td>0.01</td>
<td>0.39</td>
<td>0.49</td>
</tr>
<tr>
<td>% Difference</td>
<td>-90.91</td>
<td>-98.65</td>
<td>-55.68</td>
<td>-81.23</td>
</tr>
</tbody>
</table>

Table 8: Summary of Defects Recorded

The training time is a metric of interest for individuals considering the use of the formal notation in comparison to its semi-formal notation. In this experiment, the formal training time includes the time the subjects are in the lecture style format plus the amount of time they spend on the hands-on exercise. In addition, the amount of time it takes the subjects in the experimental group to work through the tutorial for the SRRS tool is considered as formal training. The formal training time is recorded as the number of minutes of formal training per author. The informal training includes the time the subjects use to review their training material or ask questions about the notation as they write the specification units. The informal training is recorded with respect to the number of allocated ROIDs. The total training time recorded is the sum of the formal training time and the informal training time per author. The experimental results for training time are summarized in Table 9. They show an increase in both the formal and informal training time for Group 2. The % difference between Group 1 and Group 2 is a 186% increase in total training time.

<table>
<thead>
<tr>
<th></th>
<th>Formal Training Time minutes/author</th>
<th>Informal Training Time minutes/ROID</th>
<th>Total Training Time minutes/author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>420.00</td>
<td>0.32</td>
<td>448.33</td>
</tr>
<tr>
<td>Group 2</td>
<td>835.00</td>
<td>5.02</td>
<td>1285.00</td>
</tr>
<tr>
<td>% Difference</td>
<td>98.81</td>
<td>1468.75</td>
<td>186.62</td>
</tr>
</tbody>
</table>

Table 9: Summary of Training Time

The effort is a metric of interest for individuals considering the use of the formal notation in comparison to its semi-formal notation. The effort to write and put the specification units through a peer review is summarized in Table 10. The experimental results show a decrease in the amount of time to write and review requirements for Group 2. The % difference between Group 1 and Group 2 for the total amount of time to write and review requirements is a reduction of 39%.
Appendix C. Experimental Design and Results

<table>
<thead>
<tr>
<th></th>
<th>Average time to write per ROID minutes</th>
<th>Average time to review and correct per ROID minutes</th>
<th>Average total time per ROID minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>17.58</td>
<td>15.28</td>
<td>32.86</td>
</tr>
<tr>
<td>Group 2</td>
<td>10.58</td>
<td>9.42</td>
<td>20.00</td>
</tr>
<tr>
<td>% Difference</td>
<td>-39.82</td>
<td>-38.35</td>
<td>-39.14</td>
</tr>
</tbody>
</table>

Table 10: Summary of Writing and Reviewing Time

Statistical Tests

The statistical tests done on the experimental data include the t-test, paired t-test, and P-value calculations [Mon97, Lee00]. The statistics are calculated using the statistical package Minitab. Table 11 summarizes the experimental data for testing the hypothesis for the total number of defects/ROID. Table 12 summarizes the experimental data for testing the hypothesis for the training time for the subjects. Table 13 summarizes the experimental data for testing the hypothesis for total amount of time to write, review and correct the specifications. The t-test and p value calculations for the data are presented after each table.

Defect Rate

<table>
<thead>
<tr>
<th>Specification Set</th>
<th>Group 1 (defects/ROID)</th>
<th>Group 2 (defects/ROID)</th>
<th>Difference (defects/ROID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.617</td>
<td>0.733</td>
<td>2.884</td>
</tr>
<tr>
<td>2</td>
<td>1.767</td>
<td>0.434</td>
<td>1.333</td>
</tr>
<tr>
<td>3</td>
<td>2.827</td>
<td>0.400</td>
<td>2.427</td>
</tr>
</tbody>
</table>

Table 11: Experimental Data: Total number of defects per ROID

A paired t-test is used to assess the difference across the notations.
Appendix C. Experimental Design and Results

Paired T Test and C1: C2, C1

Paired T for C2-C1

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>3</td>
<td>0.582</td>
<td>0.1823</td>
<td>0.106</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>2.373</td>
<td>0.797</td>
<td>0.536</td>
</tr>
<tr>
<td>Difference</td>
<td>3</td>
<td>-2.215</td>
<td>0.928</td>
<td>0.460</td>
</tr>
</tbody>
</table>

95% upper bound for mean distance: -0.871

T-test of mean = 0 (vs >): T-value = -4.81 P-value = 0.020

Training Time

<table>
<thead>
<tr>
<th>Specification Set</th>
<th>Group 1 (minutes)</th>
<th>Group 2 (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>445</td>
<td>1090</td>
</tr>
<tr>
<td>2</td>
<td>460</td>
<td>1055</td>
</tr>
<tr>
<td>3</td>
<td>440</td>
<td>1710</td>
</tr>
</tbody>
</table>

Table 12: Experimental Data: Total training time

Test statistics:

2-Sample T for C1 vs C2

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>3</td>
<td>1245</td>
<td>368</td>
<td>213</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>448.3</td>
<td>10.4</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Difference = μ1 - μ2

Estimate for difference: 837
Appendix C. Experimental Design and Results

95% lower bound for distance: 383

T-test of mean = 0 (vs >): T-value = 3.93 P-value = 0.009 DF = 4

Both use pooled StdDev = 261

<table>
<thead>
<tr>
<th>Specification Set</th>
<th>Group 1 (minutes/ROID)</th>
<th>Group 2 (minutes/ROID)</th>
<th>difference (minutes/ROID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>57.833</td>
<td>29.833</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>25.000</td>
<td>18.384</td>
<td>6.616</td>
</tr>
<tr>
<td>3</td>
<td>26.318</td>
<td>16.091</td>
<td>10.227</td>
</tr>
</tbody>
</table>

Table 13: Experimental Data: Writing, reviewing, and correction time per ROID

The paired t-test is used here to assess the difference across the notations.

Paired T Test and C1: C2, C1

Paired T for C2-C1

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>3</td>
<td>21.4</td>
<td>7.4</td>
<td>4.3</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>36.4</td>
<td>18.6</td>
<td>10.7</td>
</tr>
<tr>
<td>Difference</td>
<td>3</td>
<td>-14.95</td>
<td>11.45</td>
<td>6.61</td>
</tr>
</tbody>
</table>

95% upper bound for mean distance: 4.35

T-test of mean = 0 (vs <): T-value = -2.26 P-value = 0.076

1.2.3 Sources of Error

As the data is recorded and analyzed manually, the following types of error may have
Appendix C. Experimental Design and Results

occurred:

1. incorrect interpretation of SLS ROID
2. missing or incorrect identification of defect in peer review
3. incorrect or missing time record
4. mathematical errors in the analysis of the data
5. bias introduced in training session
6. time delay between running the control group and the experimental group

The subjects for the control and experimental groups are not randomly selected; the variation in the subjects is not an objective measure of the population of undergraduate students. The assumption in the experimental design is that the students selected are as similar as possible making the blocking appropriate.

1.2.4 Conclusions

The formal and informal training time both increased as expected for the Group using the formal notation in comparison to the group using the semi-formal notation. The additional burden of working through a tutorial, learning how the tool support works, and understanding the organization of the user manual all contribute to this increase. There is sufficient evidence in the results to reject the null hypothesis.

The large increase in the informal training time is an interesting and unanticipated result. An explanation for this large increase is the additional complexity of having a concrete syntax and the tool support for the notation. Since the syntax must be conformed to and the validation checks all passed, the authors of the requirements may need to check the user manual, training material, and the tutorial documents more frequently as they work on the specifications.

The experimental results show a reduction in the number of defects detected in the peer review process. The concrete syntax in addition to the tool support that enforces the syntax and provides validation checks on the specification units allow the author to check their specifications before submitting them for review. Since only specifications that have a clean validation run (no errors are reported) with the tool support are allowed to go through the peer review process in the experiment, the reviewers receive a version that has had at least some of the defects removed. As a result the specification units have a lower detected defect rate and have a higher quality. The benefit of removing the machine detectable defects before the peer review is that the reviewers can concentrate on finding defects in the specification that need human analysis. There is sufficient evidence in the results to support the rejection of the null hypothesis.

The experimental results also indicate there is a reduction in effort to write, review, and correct the specification units when using the formal notation in comparison to the semi-formal notation. The reduction can be attributed to the readability of the formal notation and the tool support. The reduced writing, review, and correction time indicates that the formal notation is at least as readable as the semi-formal notation. If the formal notation is not as readable, the time to
write, review, and correct is expected to exceed the time when the semi-formal notation is used. The tool support allows the author to obtain feedback on syntax, type, and a small number of analysis defects as the specification is being written. These defects can be removed before the specification is submitted for peer review. This reduces the number of defects in the specification making the remaining defects simpler and faster to identify in the peer review. A reduction in time is a benefit to the project, as it reduces the cost of developing the requirements specification. Although the results look promising, there is not sufficient evidence in the data to support the rejection of the null hypothesis.

With these experimental results an estimate of the training time for a project can be made. Given the notation proposed, number of authors, and number of allocated ROIDs to be written.
Appendix D. Library Example

The 18 requirements specifications provided in this Appendix have been written by the students in the experiment. They are based on a RFP for an automated on-line library system [Cor87].

Title: \Create Booking\

Overview:

The Create_Booking specification unit describes the processing performed when an authorized staff member requests to create a booking for a borrower.

Stimuli:

1) The Create_Booking shall satisfy the requirements described below upon receipt of a [Booking Function Request] from:
   a) Operator <Booking Function Request>.

2) The Create_Booking shall satisfy the requirements described below upon receipt of a [Booking Item Request] from:
   a) Operator <Booking Item Request>.

3) The Create_Booking shall satisfy the requirements described below upon receipt of a [Password Request] from:
   a) Operator <Password Request>.

Responses:

1) As specified by the requirements described below, the Create_Booking shall return a [rejection with reason] to:
   a) Operator <rejection with reason>.

2) As specified by the requirements described below, the Create_Booking shall return a [acceptance] to:
Operator <acceptance>.

As specified by the requirements described below, the Create_Booking shall commit a [commit response] as:

a) <input data attributes> to <booking transaction data>.

Requirements:

% The idea here is to allow the operator to enter a borrower number once, using %
% a Booking Function Request and to then allow items to be booked using multiple %
% Booking Item Requests. This means that each non-rejected item will update the %
% data but the data will be committed only when the password is entered. This %
% does assume that a incorrect password will not require the data to be rolled- %
% back, or conversely that the data will not actually be written with an %
% “update”. %

1) Upon receipt of a [Booking Function Request],
if {not {all of the following conditions are true:
   a) {the borrower is not a delinquent};
   b) {the unique borrower number is in the borrower list};
   c) {the Password is valid}}},
   the Create_Booking shall return a [rejection with reason].

2) Upon receipt of a [Booking Item Request],
if {not {all of the following conditions are true:
   a) {the unique borrower number is in the borrower list};
   b) {the unique item identifier is in the item list};
   c) {the branch is open on pickup date};
Appendix D. Library Example

d) {the address is valid};
e) {the item is available to be booked at this time};
f) {the item does not have status lost, missing, damaged, claims returned, or hold};
g) {the item is designated bookable};
h) {the Password is valid}}},

the Create_Booking shall return a [rejection with reason].

3) Upon receipt of a [Password Request],
   if {not {the Password is valid}}, the Create_Booking shall return a [rejection with reason].

4) Upon receipt of a [Booking Function Request],
   if {not {all of the following conditions are true:
   a) {the Password is valid}};
   b) {the unique borrower number is in the borrower list}}},
   the Create_Booking shall return a [rejection with reason].

5) Upon receipt of a [Booking Item Request], if {all of the following conditions are true:
   a) {the Booking Function Request was not rejected};
   b) {the Booking Item Request was not rejected};
   c) {the Password is valid}},{
   the Create_Booking shall return an [acceptance].

6) Upon receipt of a [Password Request], if {the Password is valid},
   the Create_Booking shall return an [acceptance].
7) Upon receipt of a [Booking Item Request],
if {all of the following conditions are true:
  a) {the Booking Function Request was not rejected};
  b) {the Booking Item Request was not rejected}
  c) {the Password is valid}},{
the Create_Booking shall commit a [commit response].

Declarations:
Type Names:
1) <input data attributes>.
2) <booking transaction data>.

Subtypes:
Static Associations:
1) Every <booking transaction data> is associated with
  exactly one <input data attributes>.

Constants:
Variable System Parameters:
System State Components:
1) “the unique borrower number is in the borrower list” : <bool>.
2) “the unique item identifier is in the item list” : <bool>.
3) “the branch is open on pickup date” : <bool>.
4) “the address is valid” : <bool>.
5) "the item is available to be booked at this time": <bool>.
6) "the item does not have status lost, missing, damaged, claims returned, or hold": <bool>.
7) "the item is designated bookable": <bool>.
9) "the Password is valid": <bool>.
10) "the borrower is not a delinquent": <bool>.
11) "the Booking Item Request was not rejected": <bool>.
12) "at least one Booking Item Request was not rejected": <bool>.
13) "the Booking Function Request was not rejected": <bool>.

Stimulus Response Components:
1) <Booking Function Request>: <request message>.
2) <Booking Item Request>: <request message>.
3) <Password Request>: <request message>.
4) <rejection with reason>: <response message>.
5) <acceptance>: <response message>.
6) <booking rejection>: <response message>.

Local Functions:

Local Predicates:
1) "The response times for the I/O transactions described in this specification unit shall be in Class" (<integer>) is a predicate.

External Functions:

External Predicates:
Appendix D. Library Example

Exported Functions:

Exported Predicates:

Performance:

1) {The response times for the I/O transactions described in this specification unit shall be in Class {2}}.
Title: \Delete Catalogue Item\n
Overview:

The Delete_Item specification unit describes the processing performed when an authorized staff member requests this subsystem to delete a catalogue item.

Stimuli:

1) The Delete_Item shall satisfy the requirements described below upon receipt of a [Delete Item Request] from:
   a) Operator <Delete Item Request>.

Responses:

1) As specified by the requirements described below, the Delete_Item shall return a [rejection with reason] to:
   a) Operator <rejection with reason>.

2) As specified by the requirements described below, the Delete_Item shall return an [acceptance] to:
   a) Operator <acceptance>.

3) As specified by the requirements described below, the Delete_Item shall commit a [delete item] as:
   a) <item> to <item list>.

Requirements:
Appendix D. Library Example

1) Upon receipt of a [Delete Item Request], if {not {all of the following conditions are true:
   a) {the User is a library staff member};
   b) {the Unique Item Identification number is valid}}},
the Delete_Item shall return a [rejection with reason].

2) Upon receipt of a [Delete Item Request], if {the Delete Item Request was not rejected},
the Delete_Item shall return a [acceptance].

3) Upon receipt of a [Delete Item Request], if {the Delete Item Request was not rejected},
the Delete_Item shall commit a [delete item].

Declarations:

Type Names:

1) <item>.

2) <item list>.

Subtypes:

Static Associations:

1) Every <item list> is associated with exactly one <item>.

Constants:

Variable System Parameters:

System State Components:

1) “the User is a library staff member” : <bool>.

2) “the Unique Item Identification number is valid” : <bool>. 
Appendix D. Library Example

3) "the Delete Item Request was not rejected": <bool>.

Stimulus Response Components:
1) <Delete Item Request>:<request message>.
2) <rejection with reason>:<response message>.
3) <acceptance>:<response message>.

Local Functions:
Local Predicates:
External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:
Title: \Discharge item non interLibrary\n
Overview:

The Discharge_Item specification unit describes the processing performed when an authorized staff member requests this subsystem to discharge an Item. Staff may search for the item to be discharged and discharge it w/o the item.

Stimuli:
1) The Discharge_Item shall satisfy the requirements described below upon receipt of a [Discharge Known Item Request] from:
   a) Operator <Discharge Known Item Request>.

2) The Discharge_Item shall satisfy the requirements described below upon receipt of a ['Sounds Like’ Request] from:
   a) Operator <'Sounds Like’ Request>.

Responses:
1) As specified by the requirements described below, the Discharge_Item shall return a [rejection with reason] to:
   a) Operator <rejection with reason>.

2) As specified by the requirements described below, the Discharge_Item shall return a [acceptance] to:
   a) Operator <acceptance>.

3) As specified by the requirements described below, the Discharge_Item shall
commit a [commit response] as:
   a) <input data attributes> to <discharge item>.

4) As specified by the requirements described below, the Discharge_Item shall send an [selection list] to:
   a) Operator <selection list>.

5) As specified by the requirements described below, the Discharge_Item shall send a [calculate fines response] to:
   a) Operator <calculate fines response>.

6) As specified by the requirements described below, the Discharge_Item shall commit a [fine] as:
   a) <fine> to <borrower's account>.

% My Vision: The operator will either enter:
% a) enter an item to discharge
% b) request a listing of items which fall under
%    some category, from which the operator will
%    either choose one and discharge it (a) again
%    or request more items from the category,
%    or revise some criteria to search.

Requirements:
1) Upon receipt of a [Discharge Known Item Request],
Appendix D. Library Example

if {not (all of the following conditions are true:

a) (the User is a library staff member);

b) (the unique item identifier is valid);

c) (the item is status is either ‘missing’ or ‘charged out’);

d) (the item is not on hold)

e) (the new status is one of ‘to be shelved’, ‘mending’,

‘binding’ or an item condition status)

f) (the transaction date is after the due date)},

the Discharge_Item shall return a [rejection with reason].

2) Upon receipt of a [Discharge Known Item Request],

if {the request was not rejected},

the Discharge_Item shall return an [acceptance].

3) Upon receipt of a [‘Sounds Like’ Request],

the Discharge_Item shall send a [selection list].

4) Upon receipt of a [Discharge Known Item Request],

if {the transaction date is after the due date},

the Discharge_Item shall:

a) send a [calculate fines response];

b) commit a [fine]. {the account shall include the new fines}.

5) Upon receipt of a [Discharge Known Item Request],

if {the request was not rejected},
the Discharge_Item shall commit a [commit response].

Declarations:

Type Names:

b) <borrower's account>.
c) <list of items>.
d) <input data attributes>.
e) <discharge item>.
f) <fine>.

Subtypes:

Static Associations:

1) Every <borrower's account> is associated with exactly one <fine>.
2) Every <discharge item> is associated with exactly one <input data attributes>.

Constants:

Variable System Parameters:

System State Components:

1) "the User is a library staff member" : <bool>.
2) "the unique item identifier is valid" : <bool>.
3) "the item is status is either 'missing' or 'charged out'" : <bool>.
4) "the item is not on hold" : <bool>.
5) "the new status is one of 'to be shelved', 'mending', 'binding' or an item condition status" : <bool>.
6) "the transaction date is after the due date" : <bool>.
7) "the account shall include the new fines" : <bool>.
Appendix D. Library Example

8) "the request was not rejected" : <bool>.

Stimulus Response Components:
1) <Discharge Known Item Request> : <request message>.
2) <'Sounds Like' Request> : <request message>.
3) <calculate fines response> : <response message>.
3) <borrower's account response> : <response message>.
3) <rejection with reason> : <response message>.
4) <acceptance> : <response message>.
5) <selection list> : <response message>.

Local Functions:
Local Predicates:
1) "The response times for the I/O transactions described in this specification unit shall be in Class" (<integer>) is a predicate.

External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:
1) {The response times for the I/O transactions described in this specification unit shall be in Class {2}}.
Title: Modify Borrower

Overview:
The Modify_Borrower specification unit describes the processing performed when an authorized staff member requests to modify the Borrower's personal or library related information on the system.

Stimuli:
1) The Modify_Borrower shall satisfy the requirements described below upon receipt of a [Modify Borrower Request] from:
   a) Operator <Modify Borrower Request>.

2) The Modify_Borrower shall satisfy the requirements described below upon receipt of a [Updated Borrower Request] from:
   a) Operator <Updated Borrower Request>.

3) The Modify_Borrower shall satisfy the requirements described below upon receipt of a [Delete Borrower Request] from:
   a) Operator <Delete Borrower Request>.

Responses:
1) As specified by the requirements described below, the Modify_Borrower shall return a [rejection with reason] to:
   a) Operator <rejection with reason>.

2) As specified by the requirements described below, the Modify_Borrower shall return an [acceptance] to:
Appendix D. Library Example

a) Operator <acceptance>.

3) As specified by the requirements described below, the Modify_Borrower shall send a [Borrower info] to:
   a) Operator <Borrower info>.

4) As specified by the requirements described below, the Modify_Borrower shall commit a [Borrower info update] as:
   a) <Borrower data> to <Borrower DB>.

5) As specified by the requirements described below, the Modify_Borrower shall commit an [option used] as:
   a) <plus one> to <Modify Borrower stats>.

Requirements:

% The plan: The operator will request "<Modify Borrower Request" with a borrower ID etc. The Modify_Borrower system will respond by sending the operator the existing record. This record will be updated by the operator and then returned via the "Updated Borrower Request". The Modify_Borrower system then updates the records.

The operator also has the option of deleting a record, via the "Delete Borrower Request". The Operator must first request to modify the borrower info, get the account, just like modify. Then s/he may select delete. This ensures that the operator will see which account is being deleted, and presumably any fines, etc.
1) Upon receipt of a [Modify Borrower Request],
if {not {all of the following conditions are true:
a) {the User is a library staff member};
b) {the unique borrower identifier is valid}}},
the Modify_Borrower shall return a [rejection with reason].

2) Upon receipt of a [Updated Borrower Request],
if {not {all of the following conditions are true:
a) {the User is a library staff member};
b) {the unique borrower identifier is valid};
c) {the fields in the new record are all valid}}},
the Modify_Borrower shall return a [rejection with reason].

3) Upon receipt of a [Delete Borrower Request],
if {not {all of the following conditions are true:
a) {the User is a library staff member};
b) {the unique borrower identifier is valid}}},
the Modify_Borrower shall return a [rejection with reason].

4) Upon receipt of a [Modify Borrower Request],
if {the Modify Borrower Request was not rejected},
the Modify_Borrower shall return a [acceptance].
5) Upon receipt of a [Updated Borrower Request],
if {the Updated Borrower Request was not rejected},
the Modify_Borrower shall return a [acceptance].

6) Upon receipt of a [Delete Borrower Request],
if {the Delete Borrower Request was not rejected},
the Modify_Borrower shall return a [acceptance].

7) Upon receipt of a [Modify Borrower Request],
if {the Modify Borrower Request was not rejected},
the Modify_Borrower shall send a [Borrower info].

8) Upon receipt of a [Updated Borrower Request],
if {the Updated Borrower Request was not rejected},
the Modify_Borrower shall:
a) commit a [Borrower info update];
b) commit a [option used].

9) Upon receipt of a [Delete Borrower Request],
if {the Delete Borrower Request was not rejected},
the Modify_Borrower shall:
a) commit a [Borrower info update];
b) commit a [option used].

Declarations:
Type Names:
Appendix D. Library Example

1) <Borrower DB>.
2) <Borrower data>.
3) <plus one>.
4) <Modify Borrower stats>.

Subtypes:

Static Associations:
1) Every <Borrower DB> is associated with exactly one <Borrower data>.
2) Every <Modify Borrower stats> is associated with exactly one <plus one>.

Constants:

Variable System Parameters:

System State Components:
1) “the User is a library staff member” : <bool>.
2) “the unique borrower identifier is valid” : <bool>.
3) “the fields in the new record are all valid” : <bool>.
4) “the Modify Borrower Request was not rejected” : <bool>.
5) “the Updated Borrower Request was not rejected” : <bool>.
6) “the Delete Borrower Request was not rejected” : <bool>.

Stimulus Response Components:
1) <Modify Borrower Request>:<request message>.
2) <Updated Borrower Request>:<request message>.
3) <Delete Borrower Request>:<request message>.
4) <rejection with reason>:<response message>.
Appendix D. Library Example

5) <borrower’s record>: <response message>.
6) <acceptance>: <response message>.
7) <Borrower info>: <response message>.

Local Functions:
Local Predicates:
External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:
Title: Pay-for-use Database Access Requests

Overview:
The Pay_Requests specification unit describes the processing performed when a user accesses the following “pay-for-view” databases:

a) The Mega-database;
b) The Everything for Arts database;
c) The Computing database;
d) The Commerce reference database.

Stimuli:
1) The Pay_Requests shall satisfy the requirements described below upon receipt of a [payment request] from:
   a) Operator <payment request>.

2) The Pay_Requests shall satisfy the requirements described below upon receipt of a [DB Query] from:
   a) Operator <DB Query>.

Responses:
1) As specified by the requirements described below, the Pay_Requests shall return a [rejection with reason] to:
   a) Operator <rejection with reason>.

2) As specified by the requirements described below, the Pay_Requests shall return an [acceptance] to:
Appendix D. Library Example

a) Operator <acceptance>.

3) As specified by the requirements described below, the Pay_Requests shall send a [Full DB Query Response] to:
   a) Operator <Query results>.

Requirements:
% So this is the deal:
1) The user will enter a DB query.
2) The system will respond with an acceptance.
3) The user will enter credit card/other payment method.
4) The system will return the results of the query if payment is OK.
%

1) Upon receipt of a [DB Query],
   if {not {all of the following conditions are true:
   {the DB Query makes sense};
   {the DB Query is for an allowable database}}},
   the Pay_Requests shall return an [rejection with reason].

2) Upon receipt of a [payment request],
   if {not {all of the following conditions are true:
   {the payment method is OK};
   {the DB Query was not rejected}}},
   the Pay_Requests shall return an [rejection with reason].
3) Upon receipt of a [DB Query],
if {the DB Query makes sense},
the Pay_Requests shall return an [acceptance].

4) Upon receipt of a [payment request],
if {all of the following conditions are true:
{the payment request not rejected};
{the DB Query was not rejected}},
the Pay_Requests shall return an [acceptance].

5) Upon receipt of a [payment request],
if {all of the following conditions are true:
{the payment request not rejected};
{the DB Query was not rejected}},
the Pay_Requests shall send a [Full DB Query Response].

Declarations:
Type Names:
Subtypes:
Static Associations:
Constants:
Variable System Parameters:
System State Components:
1) "the payment method is OK" : <bool>. 
Appendix D. Library Example

2) “the DB Query makes sense” : <bool>.
3) “the payment request not rejected” : <bool>.
4) “the DB Query was not rejected” : <bool>.
5) “the DB Query is for an allowable database” : <bool>.

Stimulus Response Components:
1) <payment request>:<request message>.
2) <DB Query>:<request message>.
3) <rejection with reason>:<response message>.
4) <acceptance>:<response message>.
5) <Full DB Query Response> :<response message>.
6) <Query results>:<response message>.

Local Functions:
Local Predicates:
External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:
Title: \Receive Serial\  

Overview:  
The Receive_Serial specification unit describes the processing performed when an authorized staff member requests this subsystem to Receive_Serial. As per the ROIDS, staff may:  
1) record receipt of issues.  
2) generate claims notices.  
3) generate cancellation notices.  
4) generate return notices.  
5) print serial request/claim notice/cancellation request/return notice.  
6) email serial request/claim notice/cancellation request/return notice.

Stimuli:  
1) The Receive_Serial shall satisfy the requirements described below upon receipt of a [Record Receipt request] from:  
a) Operator <Record Receipt request>.

2) The Receive_Serial shall satisfy the requirements described below upon receipt of a [Claims Notice request] from:  
a) Operator <Claims notice request>.

3) The Receive_Serial shall satisfy the requirements described below upon receipt of a [Cancellation Notice request] from:  
a) Operator <Cancellation Notice request>.
Appendix D. Library Example

4) The Receive_Serial shall satisfy the requirements described below upon receipt of a [Return Notice request] from:
   a) Operator <Return Notice request>.

Responses:
1) As specified by the requirements described below, the Receive_Serial shall return a [rejection with reason] to:
   a) Operator <rejection with reason>.

2) As specified by the requirements described below, the Receive_Serial shall return a [acceptance] to:
   a) Operator <acceptance>.

4) As specified by the requirements described below, the Receive_Serial shall send a [Record Receipt response] to:
   a) Operator <Record Receipt>.

5) As specified by the requirements described below, the Receive_Serial shall send a [Claims Notice response] to:
   a) Operator <Claims Notice>.

6) As specified by the requirements described below, the Receive_Serial shall send a [Cancellation Notice response] to:
   a) Operator <Cancellation Notice>. 
7) As specified by the requirements described below, the Receive_Serial shall send a [Return Notice response] to:
   a) Operator <Return Notice>.

8) As specified by the requirements described below, the Receive_Serial shall send a [print Receipt] to:
   b) Operator <print Receipt>.

9) As specified by the requirements described below, the Receive_Serial shall send a [print Claims Notice] to:
   a) Operator <print Claims Notice>.

10) As specified by the requirements described below, the Receive_Serial shall send a [print Cancellation Notice] to:
    a) Operator <print Cancellation Notice>.

11) As specified by the requirements described below, the Receive_Serial shall send a [print Return Notice] to:
    a) Printer <print Return Notice>.

12) As specified by the requirements described below, the Receive_Serial shall commit a [Receipt] as:
    a) <the Receipt> to <Receipt list>.

Requirements:

1) Upon receipt of a [Record Receipt request],
Appendix D. Library Example

if {not {all of the following conditions are true:
  a) {the User is a staff member};
  b) {the Receipt is valid}}},
the Receive_Serial shall return a [rejection with reason].

2) Upon receipt of a [Claims Notice request],
if {not {all of the following conditions are true:
  a) {the User is a staff member};
  b) {the claim notice period is over}}},
the Receive_Serial shall return a [rejection with reason].

3) Upon receipt of a [Cancellation Notice request],
if {not {all of the following conditions are true:
  a) {the User is a staff member};
  b) {the serial code is valid}}},
the Receive_Serial shall return a [rejection with reason].

4) Upon receipt of a [Return Notice request],
if {not {all of the following conditions are true:
  a) {the User is a staff member};
  b) {the serial code is valid}}},
the Receive_Serial shall return a [rejection with reason].

5) Upon receipt of a [Record Receipt request],
if {the Record Receipt request was not rejected},
the Receive_Serial shall return an [acceptance].

6) Upon receipt of a [Claims Notice request],
if {the Claims Notice request was not rejected},
the Receive_Serial shall return an [acceptance].

7) Upon receipt of a [Cancellation Notice request],
if {the Cancellation Notice request was not rejected},
the Receive_Serial shall return an [acceptance].

8) Upon receipt of a [Return Notice request],
if {the Return Notice request was not rejected},
the Receive_Serial shall return an [acceptance].

9) Upon receipt of a [Record Receipt request],
if {the Record Receipt request was not rejected},
the Receive_Serial shall commit a [Receipt].

10) Upon receipt of a [Claims Notice request],
if {the Claims Notice request was not rejected},
the Receive_Serial shall send a [Claims Notice response].

11) Upon receipt of a [Cancellation Notice request],
if {the Cancellation Notice request was not rejected},
the Receive_Serial shall send a [Cancellation Notice response].
12) Upon receipt of a [Return Notice request],
if {the Return Notice request was not rejected},
the Receive_Serial shall send an [Return Notice response].

13) Upon receipt of a [Record Receipt request],
if {all of the following conditions are true:
  a) {the Record Receipt request was not rejected};
  b) {a Record Receipt print was requested}},
the Receive_Serial shall send a [print Receipt].

14) Upon receipt of a [Claims Notice request],
if {all of the following conditions are true:
  a) {the Claims Notice request was not rejected};
  b) {a Claims Notice print was requested}},
the Receive_Serial shall send a [print Claims Notice].

15) Upon receipt of a [Cancellation Notice request],
if {all of the following conditions are true:
  a) {the Cancellation Notice request was not rejected};
  b) {a Cancellation Notice print was requested}},
the Receive_Serial shall send a [print Cancellation Notice].

16) Upon receipt of a [Return Notice request],
if {all of the following conditions are true:
a) {the Return Notice request was not rejected};
b) {a Return Notice print was requested}},
the Receive_Serial shall send a [print Return Notice].

Declarations:
Type Names:
1) <the Receipt>.
2) <Receipt list>.

Subtypes:
Static Associations:
1) Every <Receipt list> is associated with exactly one <the Receipt>.

Constants:
Variable System Parameters:
System State Components:
1) “the User is a staff member” : <bool>.
2) “the Receipt is valid” : <bool>.
3) “the serial code is valid” : <bool>.
4) “the Record Receipt request was not rejected” : <bool>.
5) “the Claims Notice request was not rejected” : <bool>.
6) “the Cancellation Notice request was not rejected” : <bool>.
7) “the Return Notice request was not rejected” : <bool>.
8) “a Record Receipt print was requested” : <bool>.
9) “a Claims Notice print was requested” : <bool>.
Appendix D. Library Example

11) “a Cancellation Notice print was requested” : <bool>.
12) “a Return Notice print was requested” : <bool>.
13) “the claim notice period is over” : <bool>.

Stimulus Response Components:
1) <Record Receipt request>:<request message>.
1) <Claims notice request>:<request message>.
1) <Cancellation Notice request>:<request message>.
1) <Return Notice request>:<request message>.
1) <rejection with reason>:<response message>.
1) <acceptance>:<response message>.
1) <Record Receipt response>:<response message>.
1) <Record Receipt>:<response message>.
2) <Claims Notice>:<response message>.
4) <Return Notice>:<response message>.
5) <print Receipt>:<response message>.
6) <print Claims Notice>:<response message>.
7) <print Cancellation Notice>:<response message>.
8) <print Return Notice>:<response message>.
1) <Receipt>:<response message>.
Appendix D. Library Example

Local Functions:
Local Predicates:

External Functions:
External Predicates:

Exported Functions:
Exported Predicates:

Performance:
Title: \ Create Acquisition Plan\

Overview:

The Create Acquisition Plan specification unit describes the processing performed when an operator requests to create an acquisition plan.

If the operator is an authorized library staff member with acquisition capabilities, then the operator is allowed to create an acquisition plan.

Otherwise, the request is rejected.

Stimuli:

1) The IOLS shall satisfy the requirements described below upon receipt of a [create acquisition plan request] from:
   a) Operator <create acquisition plan request>.

2) The IOLS shall satisfy the requirements described below upon receipt of a [print note request] from:
   a) Operator <print note request>.

Responses:

1) As specified by the requirements described below, the IOLS shall return a [rejection with reason] to:
   a) Operator <create acquisition plan rejection>.

2) As specified by the requirements described below, the IOLS shall return an [acceptance] to:
   a) Operator <acceptance>.
3) As specified by the requirements described below, the IOLS shall send a [acquisition plan response] to:
   a) Operator <create acquisition plan response>.

4) As specified by the requirements described below, the IOLS shall commit a [list of items] as:
   a) <list of items> to <acquisition plan>.

5) As specified by the requirements described below, the IOLS shall commit a [cost of items calculated using the discount matrix] as:
   a) <cost of items calculated using the discount matrix>.

6) As specified by the requirements described below, the IOLS shall commit a [sorted acquisition plan] as:
   a) <sorted acquisition plan> to <acquisition plan>.

7) As specified by the requirements described below, the IOLS shall send a [print note response] to:
   a) Printer <print note response>.

Requirements:
1) Upon receipt of a [create acquisition plan request],
   if {the operator number is not in the authorized library staff member’s list},
   the IOLS shall return a [rejection with reason].
2) Upon receipt of a [create acquisition plan request], if {the request is not rejected}, the IOLS shall return an [acceptance].

3) Upon receipt of a [create acquisition plan request], if {the request is not rejected}, the IOLS shall:
   a) send a [acquisition plan response];
   b) commit a [list of items];
      c) commit a [cost of items calculated using the discount matrix];
      d) commit a [sorted acquisition plan].

4) Upon receipt of a [print note request], the IOLS shall send
   a [print note response].

Declarations:

Type Names:
   1) <list of items>.
   2) <acquisition plan>.
   3) <cost of items calculated using the discount matrix>.
   4) <sorted acquisition plan>.
   5) <purchase order>.
   6) <jobber/publisher>.
   7) <number of copies ordered>.
   8) <list price>.
   9) <order type (rush, normal)>.
10) < payment type (bill, prepayment)>.

11) <title>.

12) <author>.

13) <copyright date>.

14) <OCLC/RIN>.


16) <ISSN>.

Subtypes:

Static Associations:

1) Every <acquisition plan> is associated with exactly one <list of items>.

2) Every <new items selection response> is associated with:
   a) exactly one <jobber/publisher>;
   b) exactly one <number of copies ordered>;
   c) exactly one <list price>;
   d) exactly one <order type (rush, normal)>;
   e) exactly one < payment type (bill, prepayment)>;
   f) exactly one <title>;
   g) exactly one <author>;
   h) exactly one <copyright date>;
   i) exactly one <OCLC/RIN>;
   j) exactly one <ISBN>;
   k) exactly one <ISSN>.

3) Every <acquisition plan> is associated with exactly one <sorted acquisition
plan).

Constants:

Variable System Parameters:

System State Components:

1) "the request is not rejected":<bool>.

2) "the operator number is not in the authorized library staff member’s list":<bool>.

Stimulus Response Components:

1) <create acquisition plan request>:<request message>.

2) <create acquisition plan rejection>:<response message>.

3) <acceptance>:<response message>.

4) <create acquisition plan response>:<response message>.

5) <new items selection response>:<response message>.

6) <acquisition plan response>:<response message>.

7) <print note response>:<response message>.

8) <print note request>:<request message>.

Local Functions:

Local Predicates:

External Functions:

External Predicates:

Exported Functions:

Exported Predicates:

Performance:
Appendix D. Library Example

Title: \Charge Item (interlibrary)\

Overview:
The Charge Item (interlibrary) specification unit describes the processing performed upon receipt of a request to charge an item from another library. If the item exists in the catalogue, item is classified as circulating, item is not on hold or the item is on hold for the borrower, item’s status is “on shelf”, the requesting library has an interlibrary loan agreement with MegaLibrary, borrower’s account exists, and borrower’s account status is “active”, then the librarian can charge an item to another library. Otherwise, the request is rejected.

Stimuli:
1) The IOLS shall satisfy the requirements described below upon receipt of a [charge item request] from:
   a) Operator <charge item request>.

2) The IOLS shall satisfy the requirements described below upon receipt of a [change loan period request] from:
   a) Operator <change loan period request>.

Responses:
1) As specified by the requirements described below, the IOLS shall return a [rejection with reason] to:
   a) Operator <rejection response>.
2) As specified by the requirements described below, the IOLS shall return an [acceptance] to:
   a) Operator <acceptance>.

3) As specified by the requirements described below, the IOLS shall send a [charge item response] to:
   a) Operator <charge item response>.

4) As specified by the requirements described below, the IOLS shall commit a [charged item] as:
   a) <charged item> to <borrower's account number>.

5) As specified by the requirements described below, the IOLS shall send a [change loan period response] to:
   a) Operator <change loan period response>.

6) As specified by the requirements described below, the IOLS shall commit a [loan period] as:
   a) <loan period> to <charged item identifier>.

Requirements:
1) Upon receipt of a [charge item request],
   if {one or more of the following conditions are true:
   a) {the item does not exist in the catalogue};
   b) {the item is not classified as circulating};
Appendix D. Library Example

c) { the item is on hold or the item is not on hold for the borrower }

d) { the item's status is not on shelf }

e) { the requesting library does not have an interlibrary loan agreement with MegaLibrary }

f) { borrower's account does not exist }

g) { borrower's account status is not active }

h) { the requested library is not TechnoLibrary, Artsy-Library, or Commerce-Library }

, the IOLS shall return a [rejection with reason].

2) Upon receipt of a [change loan period request],

if {one or more of the following conditions are true:

a) { the operator is not an authorized library staff }

b) { the due date occurs when the branch is closed }

, the IOLS shall return a [rejection with reason].

3) Upon receipt of a [charge item request], if {the request is not rejected},

the IOLS shall return an [acceptance].

4) Upon receipt of a [change loan period request],

if {the request is not rejected}, the IOLS shall return an [acceptance].

5) Upon receipt of a [charge item request],

if {the request is not rejected}, the IOLS shall:

a) send a [charge item response];

b) commit a [charged item].
6) Upon receipt of a [change loan period request], if {the request is not rejected}, the IOLS shall:
   a) send a [change loan period response];
   b) commit a [loan period].

Declarations:

Type Names:
1) <charged item>.
2) <loan period>.
3) <borrower’s account number>.
4) <charged item identifier>.

Subtypes:

Static Associations:
1) Every <borrower’s account number> is associated with exactly one <charged item>.
2) Every <charged item> is associated with exactly one <charged item identifier>.
3) Every <charged item identifier> is associated with exactly one <loan period>.

Constants:

Variable System Parameters:

System State Components:
Appendix D. Library Example

1) “the request is not rejected”:<bool>.
2) “the operator is not an authorized library staff”:<bool>.
3) “the item does not exist in the catalogue”:<bool>.
4) “the item is not classified as circulating”:<bool>.
5) “the item is on hold or the item is not on hold for the borrower”:<bool>.
6) “the item’s status is not on shelf”:<bool>.
7) “the requesting library does not have an interlibrary loan agreement with MegaLibrary”:<bool>.
8) “borrower’s account does not exist”:<bool>.
9) “borrower’s account status is not active”:<bool>.
10) “the requested library is not TechnoLibrary, Artsy-Library, or Commerce-Library”:<bool>.
11) “the due date occurs when the branch is closed”:<bool>.

Stimulus Response Components:
1) <charge item request>:<request message>.
2) <change loan period request>:<request message>.
3) <rejection response>:<response message>.
4) <charge item response>:<response message>.
5) <change loan period response>:<response message>.
6) <acceptance>:<response message>.

Local Functions:

Local Predicates:
1) “The response times for the I/O transactions described in this specification
unit shall be in Class” (<integer>) is a predicate.

2) “The interlibrary loan period has a default value of” (<integer>) “weeks” is a predicate.

External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:

1) {The response times for the I/O transactions described in this specification unit shall be in Class {2}}.
Title: Add Catalog Item

Overview:

The Add Catalog Item specification unit describes the processing performed upon receipt of a request to add catalog item. If the operator is a library staff, the operator is allowed to add catalogue item. Otherwise, the request is rejected.

Stimuli:

1) The IOLS shall satisfy the requirements described below upon receipt of a [add catalog item request] from:
   a) Operator <add catalogue item request>.

Responses:

1) As specified by the requirements described below, the IOLS shall return a [rejection with reason] to:
   a) Operator <add catalogue item rejection>.

2) As specified by the requirements described below, the IOLS shall return an [acceptance] to:
   a) Operator <acceptance>.

3) As specified by the requirements described below, the IOLS shall commit an [item] as:
   a) <item> to <catalog>.
Appendix D. Library Example

Requirements:

1) Upon receipt of a [add catalog item request],

   if { one or more of the following conditions are true:

      a) { the operator is not a library staff};

      b) { the item is not from the tape of MARC formatted records

         or is not entered at a MS} }

   , the IOLS shall return a [rejection with reason].

2) Upon receipt of a [add catalog item request], if {the request is not

    rejected}, the IOLS shall return an [acceptance].

3) Upon receipt of a [add catalog item request], if {the request is not

    rejected}, the IOLS shall

    commit an [item].

Declarations:

Type Names:

1) <item>.

2) <catalog>.

3) set of <item id>.

4) <author>.

5) <title>.

6) <OCLC no>.

7) <ISBN>.

8) <ISSN>.
9) <added entries>.
10) <subject headings>.

Subtypes:

Static Associations:
1) Every <catalog> is associated with exactly one <item>.
2) Every <item> is associated with one or more <item id>.
3) Every <item> is associated with:
   a) exactly one <author>;
   b) exactly one <title>;
   c) exactly one <OCLC no>;
   d) exactly one <ISBN>;
   e) exactly one <added entries>;
   f) exactly one <subject headings>.

Constants:

Variable System Parameters:

System State Components:
1) “the request is not rejected”:<bool>.
2) “the operator is not a library staff”:<bool>.
3) “the item is not from the tape of MARC formatted records
   or is not entered at a MS”:<bool>.

Stimulus Response Components:
1) <add catalogue item request>:<request message>.
Appendix D. Library Example

2) <add catalogue item rejection>:<response message>.
3) <acceptance>:<response message>.

Local Functions:
Local Predicates:
External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:
Title: \Modify Acquisition Plan\ 

Overview:

The Modify Acquisition Plan specification unit describes the processing performed when an operator requests to modify an acquisition plan.

If the operator is an authorized library staff member with the acquisition capabilities, then the operator is allowed to modify acquisition plan.

Otherwise, the request is rejected.

Stimuli:

1) The Library_System shall satisfy the requirements described below upon receipt of a [modify acquisition plan request] from:
   a) Operator <modify acquisition plan request>.

2) The Library_System shall satisfy the requirements described below upon receipt of a [print note request] from:
   a) Operator <print note request>.

Responses:

1) As specified by the requirements described below, the Library_System shall return a [rejection with reason] to:
   a) Operator <modify acquisition plan rejection>.

2) As specified by the requirements described below, the Library_System shall return an [acceptance] to:
   a) Operator <acceptance>.
3) As specified by the requirements described below, the Library_System shall send a [modify acquisition plan response] to:
   a) Operator <acquisition plan response>.

4) As specified by the requirements described below, the Library_System shall commit a [part of acquisition plan] as:
   a) <part of acquisition plan> to <acquisition plan>.

5) As specified by the requirements described below, The Library_System shall commit a [cost calculation of purchasing items using the discount matrix] as:
   a) <that cost to commit>.

6) As specified by the requirements described below, the Library_System shall commit a [sorted acquisition plan] as:
   a) <sorted acquisition plan> to <acquisition plan>.

7) As specified by the requirements described below, the Library_System shall send a [print note response] to:
   a) Printer <print note response>.

8) As specified by the requirements described below, the Library_System shall commit a [cancellation date] as:
   a) <that cancellation date to commit>.
Appendix D. Library Example

Requirements:

1) Upon receipt of a [modify acquisition plan request], if {the operator number is not in the authorized library staff member's list}, the Library_System shall return a [rejection with reason].

2) Upon receipt of a [modify acquisition plan request], if {the request is not rejected}, the Library_System shall return an [acceptance].

3) Upon receipt of a [modify acquisition plan request], if {the request is not rejected}, the Library_System shall:
   a) send a [modify acquisition plan response];
   b) commit a [part of acquisition plan]. { The committed part of acquisition plan shall include the current plan};
   c) commit a [cost calculation of purchasing items using the discount matrix];
   d) commit a [sorted acquisition plan];
   e) commit a [cancellation date].

4) Upon receipt of a [print note request], the Library_System shall send a [print note response].

Declarations:

Type Names:

1) <part of acquisition plan>.
Appendix D. Library Example

2) <acquisition plan>.
3) set of <acquisition plan id>.
4) <cost calculation of purchasing items using the discount matrix>.
5) <sorted acquisition plan>.
6) <that cancellation date to commit>.
7) <cancellation date>.
8) <purchase order>.
9) <jobber/publisher>.
10) <number of copies ordered>.
11) <list price>.
12) <order type (rush, normal)>.
13) <payment type (bill, prepayment)>.
14) <title>.
15) <author>.
16) <copyright date>.
17) <OCLC/RIN>.
19) <ISSN>.
20) <that cost to commit>.

Subtypes:

Static Associations:

1) Every <acquisition plan> is associated with exactly one <part of acquisition plan>.
2) Every <part of acquisition plan> is associated with one or more <acquisition plan id>.
3) Every <new items selection response> is associated with:
   a) exactly one <jobber/publisher>;
   b) exactly one <number of copies ordered>;
   c) exactly one <list price>;
   d) exactly one <order type (rush, normal)>;
   e) exactly one <payment type (bill, prepayment)>;
   f) exactly one <title>;
   g) exactly one <author>;
   h) exactly one <copyright date>;
   i) exactly one <OCLC/RIN>;
   j) exactly one <ISBN>;
   k) exactly one <ISSN>.

4) Every <purchase order> is associated with zero or one <cancellation date>.

5) Every <acquisition plan> is associated with zero or one <sorted acquisition plan>.

Constants:

Variable System Parameters:

System State Components:

1) “the request is not rejected”:<bool>.

2) “The committed part of acquisition plan shall include the current plan”:<bool>.

3) “the operator number is not in the authorized library staff member’s list”:<bool>.
5) "the part of acquisition plan to the acquisition plan is equal to the maximum number of parts per plan":<bool>.

Stimulus Response Components:
1) <modify acquisition plan request>:<request message>.
2) <modify acquisition plan rejection>:<response message>.
3) <acceptance>:<response message>.
4) <modify acquisition plan response>:<response message>.
5) <new items selection response>:<response message>.
6) <acquisition plan response>:<response message>.
7) <print note response>:<response message>.
8) <print note request>:<request message>.

Local Functions:

Local Predicates:
1) "The response times for the I/O transactions described in this specification unit shall be in Class" (<integer>) is a predicate.

External Functions:

External Predicates:

Exported Functions:

Exported Predicates:

Performance:
1) {The response times for the I/O transactions described in this specification unit shall be in Class {2}}.
Title: \Modify on-line adaptation data\n
Overview:

The Modify on-line adaptation data specification unit describes the processing performed when an operator requests to modify on-line adaptation data. If the operator is an authorized library staff member then the operator is allowed to modify fine, service charge, overdue notification parameters and loan period. Otherwise, the request is rejected.

Stimuli:

1) The Library_System shall satisfy the requirements described below upon receipt of a [modify adaptation data request] from:
   a) Operator <modify adaptation data request>.

Responses:

1) As specified by the requirements described below, the Library_System shall return a [rejection with reason] to:
   a) Operator <modify adaptation data rejection>.

2) As specified by the requirements described below, the Library_System shall return an [acceptance] to:
   a) Operator <acceptance>.

3) As specified by the requirements described below, the Library_System shall
commit a [part of adaptation data] as:

a) <part of adaptation data> to <adaptation data>.

Requirements:

1) Upon receipt of a [modify adaptation data request], if 
   { the operator is not the authorized library staff}, 
   the Library_System shall return a [rejection with reason].

2) Upon receipt of a [modify adaptation data request], if {the request is not rejected}, the Library_System shall return an [acceptance].

3) Upon receipt of a [modify adaptation data request], if {the request is not rejected}, the IOLS shall commit a [part of adaptation data].

Declarations:

Type Names:

1) <part of adaptation data>.
2) <adaptation data>.
3) <number of renewals>.
4) <thresholds for fines>.
5) <jobbers>.
6) <claim notice period>.
7) <cancellation notice period>.
8) <claim notice for a serial period>.
8) <max binding number of days>.
Appendix D. Library Example

9) <fine>.
10) <service charge>.
11) <overdue notification parameters>.
12) <loan period>.
13) <interlibrary loan period>.

Subtypes:

Static Associations:

1) Every <adaptation data> is associated with exactly one <part of adaptation data>.

2) Every <part of adaptation data> is associated with:
   a) exactly one <number of renewals>;
   b) exactly one <thresholds for fines>;
   c) exactly one <jobbers>;
   d) exactly one <claim notice period>;
   e) exactly one <cancellation notice period>;
   f) exactly one <claim notice for a serial period>;
   g) exactly one <max binding number of days>.

3) Every <modify adaptation data request> is associated with:
   a) zero or one <fine>;
   b) zero or one <service charge>;
   c) zero or one <overdue notification parameters>;
   d) zero or one <loan period>;
   e) zero or one <interlibrary loan period>.
Appendix D. Library Example

Constants:

Variable System Parameters:

System State Components:
1) "the request is not rejected":<bool>.
2) "the operator is not the authorized library staff":<bool>.

Stimulus Response Components:
1) <modify adaptation data request>:<request message>.
2) <modify adaptation data rejection>:<response message>.
3) <acceptance>:<response message>.

Local Functions:

Local Predicates:
1) "The response times for the I/O transactions described in this specification unit shall be in Class" (<integer>) is a predicate.
2) "The interlibrary loan period has a default value of" (<integer>) "weeks" is a predicate.
3) "The minimum number of jobbers has a default value of" (<integer>) is a predicate.
4) "The claim notice period has a default value of" (<integer>) "days" is a predicate.
5) "The cancellation notice period has a default value of" (<integer>) "days" is a predicate.
6) "The claim notice period for a serial has a default value of" (<integer>) "days" is a predicate.
Appendix D. Library Example

7) "The maximum binding number of days has a default value of" (<integer>)
   "days" is a predicate.

8) "The number of renewals has a default value of" (<integer>) "times"
   is a predicate.

External Functions:
External Predicates:
Exported Functions:
Exported Predicates:
Performance:
Title: Request Statistics on interlibrary loans

Overview:
The Receive Acquisition specification unit describes the processing performed upon receipt of a request to maintain statistics on interlibrary loans. The statistics for interlibrary loan includes number of borrowing transactions per library and number of lending transactions per library.

Stimuli:
1) The IOLS shall satisfy the requirements described below upon receipt of a [request statistics] from:
   a) Operator <request statistics>.

Responses:
2) As specified by the requirements described below, the IOLS shall send a [statistics report] to:
   a) Operator <statistics report>.

Requirements:
1) Upon receipt of a [request statistics], the IOLS shall send a [statistics report].

Declarations:
Type Names:
1) set of <number of borrowing transactions per library>.

337
Appendix D. Library Example

2) set of <number of lending transactions per library>.

Subtypes:

Static Associations:

1) Every <statistics report> is associated with:
   a) one or more <number of borrowing transactions per library>;
   b) one or more <number of lending transactions per library>.

Constants:

Variable System Parameters:

System State Components:

Stimulus Response Components:

1) <request statistics>:<request message>.
2) <statistics report>:<response message>.

Local Functions:

Local Predicates:

External Functions:

External Predicates:

Exported Functions:

Exported Predicates:

Performance:
Title: \Receive Acquisition \n
Overview:
The Receive Acquisition specification unit describes the processing performed upon receipt of a request to receive acquisition. The system provide the capability to update the status of an item on order, automatically generate a claims notice for items order, cancellation notice items on order, to record the receipt of ordered items, to generate a return notice, to update the status of items in the catalog back-ordered, pending, or cancelled; to print or e-mail claims notice, cancellation notice, receipt report, and return notice.

Stimuli:
1) The IOLS shall satisfy the requirements described below upon receipt of a [receive acquisition request] from:
a) Operator <receive acquisition request>.

2) The IOLS shall satisfy the requirements described below upon receipt of a [print request] from:
a) Operator <print request>.

Responses:
1) As specified by the requirements described below, the IOLS shall return an [acceptance] to:
a) Operator <acceptance>.
2) As specified by the requirements described below, the IOLS shall send a [receive acquisition response] to:
   a) Operator <receive acquisition response>.

3) As specified by the requirements described below, the IOLS shall update an [item's order] as:
   a) <item's order> to <acquisition>.

4) As specified by the requirements described below, the IOLS shall commit a [claims notice] as:
   a) <claims notice> to <item's order>.

5) As specified by the requirements described below, the IOLS shall commit a [cancellation notice] as:
   a) <cancellation notice> to <item's order>.

6) As specified by the requirements described below, the IOLS shall commit a [return notice] as:
   a) <return notice> to <item's order>.

7) As specified by the requirements described below, the IOLS shall commit a [receipt report] as:
   a) <receipt report> to <item's order>.
8) As specified by the requirements described below, the IOLS shall update a [status of items] as:
   a) [status of items] to [item’s order].

9) As specified by the requirements described below, the IOLS shall send a [print response] to:
   a) Printer [print response].

Requirements:
1) Upon receipt of a [receive acquisition request], the IOLS shall return an [acceptance].

2) Upon receipt of a [print request], the IOLS shall return an [acceptance].

3) Upon receipt of a [receive acquisition request], the IOLS shall:
   a) send a [receive acquisition response];
   b) update an [item’s order].

4) Upon receipt of a [receive acquisition request], if {needs a claim notice},
   the IOLS shall commit a [claims notice].

5) Upon receipt of a [receive acquisition request], if {needs a cancellation notice}, the IOLS shall commit a [cancellation notice].
6) Upon receipt of a [receive acquisition request], if {needs a return notice}, the IOLS shall commit a [return notice].

7) Upon receipt of a [receive acquisition request], if {needs a receipt report}, the IOLS shall commit a [receipt report].

8) Upon receipt of a [print request], if {needs to print cancellation notice or claims notice, or receipt report, or return notice}, the IOLS shall send a [print response].

9) Upon receipt of an [receive acquisition request], if {needs to update the status of items}, the IOLS shall update a [status of items].

Declarations:
Type Names:
1) <OO on order>.
2) <BOS back order, temporarily out of stock>.
3) <BON back order, not a stocked item>.
4) <BOWR back order, title is not available and will be shipped when ready>.
5) <BOPO back order, publisher out of stock>.
6) <RES back order received>.
7) <PEND pending, title is in the process of being shipped and billed>.
8) <OP out of print, canceled>.
9) <OD order direct, canceled>.
10) <S subscription, order direct, canceled>.
11)< PNS publisher not supplied, canceled >.
12)< TNS title not supplied, canceled >.
13)< PFR publisher failed to respond, canceled >.
14)< PC publication canceled, < PFR publisher failed to respond, canceled >.
15)< CF cannot find, canceled >.
16)< NOP not publication of publisher, canceled >.
17)< POS publisher out of stock >.
18)< TOS temporarily out of stock, canceled due to cancellation >.
19)< REC received >.
20)< claims notice >.
21)< cancellation notice >.
22)< return notice >.
23)< receipt report >.
24)< status of items >.
26)< item’s order >.
27)< acquisition >.
28) set of < item’s order id >.

Subtypes:

Static Associations:

1) Every < item’s order > is associated with:
   a) exactly one < OO on order >;
   b) exactly one < BOS back order, temporarily out of stock >;
   c) exactly one < BON back order, not a stocked item >;
d) exactly one <BOWR back order, title is not available and will be shipped when ready>;

e) exactly one <BOPO back order, publisher out of stock>;

f) exactly one <RES back order received>;

g) exactly one <PEND pending, title is in the process of being shipped and billed>;

h) exactly one < OP out of print, canceled>;

i) exactly one < OD order direct, canceled>;

j) exactly one < S subscription, order direct, canceled>;

k) exactly one < PNS publisher not supplied, canceled>;

l) exactly one < TNS title not supplied, canceled>;

m) exactly one < PFR publisher failed to respond, canceled>;

n) exactly one < PC publication canceled, < PFR publisher failed to respond, canceled>;

o) exactly one < CF cannot find, canceled>;

p) exactly one < NOP not publication of publisher, canceled>;

q) exactly one < POS publisher out of stock>;

r) exactly one <TOS temporarily out of stock, canceled due to cancellation>;

s) exactly one < REC received>.

2) Every <acquisition> is associated with exactly one <item’s order>.

3) Every <item’s order> is associated with one or more <item’s order id>.

4) Every <item’s order> is associated with exactly one <claims notice>.

5) Every <item’s order> is associated with exactly one <cancellation notice>.

6) Every <item’s order> is associated with exactly one <return notice>.
7) Every <item’s order> is associated with exactly one <receipt report>.

8) Every <item’s order> is associated with exactly one <status of items>.

Constants:

Variable System Parameters:

System State Components:

1) “needs a claim notice”:<bool>.

2) “needs a cancellation notice”:<bool>.

3) “needs a return notice”:<bool>.

4) “needs a receipt report”:<bool>.

5) “needs to print a cancellation report or claim report, or return notice, or receipt report”:<bool>.

6) “needs to update the status of items”:<bool>.

Stimulus Response Components:

1) <receive acquisition request>:<request message>.

2) <acceptance>:<response message>.

3) <receive acquisition response>:<response message>.

4) <print request>:<request message>.

5) <print response>:<response message>.

Local Functions:

Local Predicates:

1) “The response times for the I/O transactions described in this specification unit shall be in Class” (<integer>) is a predicate.

2) “The cancellation notice period has a default value of” (<integer>) “days” is a predicate.
3) “The claim notice period has a default value of” (<integer>) “days” is a predicate.

External Functions:

External Predicates:

Exported Functions:

Exported Predicates:

Performance:

1) {The response times for the I/O transactions described in this specification unit shall be in Class \{2\}.}
Title: Renew Item (with, without item) \n
Overview:

The Renew Item specification unit describes the processing performed upon receipt of a request to renew an item. If the operator is a borrower or library staff, the borrower's account exists and is active, the borrower's card is not expired; the item is on a local charge and the item is not Interlibrary loan charge, the operator is allowed to renew item. Otherwise, the request is rejected.

Stimuli:

1) The IOLS shall satisfy the requirements described below upon receipt of a [renew item request] from:
   a) Operator <renew item request>.

2) The IOLS shall satisfy the requirements described below upon receipt of a [renew loan period request] from:
   a) Operator <renew loan period request>.

Responses:

1) As specified by the requirements described below, the Library_System shall return a [rejection with reason] to:
   a) Operator <renew item rejection>.

2) As specified by the requirements described below, the Library_System shall
Appendix D. Library Example

return an [acceptance] to:

  a) Operator <acceptance>.

3) As specified by the requirements described below, the Library_System shall
   send a [renew item response] to:
   a) Operator <renew item response>.

4) As specified by the requirements described below, the Library_System shall
   commit a [renew item] as:
   a) <renew item> to <borrower’s account number>.

5) As specified by the requirements described below, the Library_System shall
   send a [renew loan period response] to:
   a) Operator <renew loan period response>.

6) As specified by the requirements described below, the Library_System shall
   commit a [loan period] as:
   a) <loan period> to <renew item identifier>;
   
   b) <loan period> to <borrower’s account number>.

Requirements:

1) Upon receipt of a [renew item request],
   if {one or more of the following conditions are true:
   a) { the operator is not a library staff or borrower};
   b) { borrower’s account doesn’t exist};
Appendix D. Library Example

c) { borrower's account is not active};

d) { the borrower has to make a payment};

e) { the item is not on a local charge }; 

f) { the item is on Interlibrary charge};

g) { internal department is not mending or binding};

h) { the item is overdue} }

, the Library_System shall return a [rejection with reason].

2) Upon receipt of a [renew loan period request],

if {one or more of the following conditions are true: 

a) { the due date occurs when the borrower's card is expired }; 

b) { the number of renewals is more than a possible limit};

c) { the due date occurs when the library is closed}}

, the Library_System shall return a [rejection with reason].

3) Upon receipt of a [renew item request], if {the request is not rejected},

the Library_System shall return an [acceptance].

4) Upon receipt of a [renew loan period request],

if {the request is not rejected}, the Library_System shall return an [acceptance].

5) Upon receipt of a [renew item request], the Library_System shall:

a) send a [renew item response];

b) commit a [renew item].
6) Upon receipt of a [renew loan period request], the Library_System shall:
   a) send a [renew loan period response];
   b) commit a [loan period].

Declarations:
Type Names:
1) <renew item>.
2) <loan period>.
3) <borrower's account number>.
4) <renew item identifier>.
5) set of <renew item id>.
5) set of <loan period id>.

Subtypes:

Static Associations:
1) Every <borrower's account number> is associated with exactly one <renew item>.
2) Every <renew item> is associated with zero or more <renew item id>.
3) Every <renew item identifier> is associated with exactly one <loan period>.
3) Every <borrower's account number> is associated with exactly one <loan period>.
4) Every <loan period> is associated with zero or more <loan period id>.

Constants:
Appendix D. Library Example

Variable System Parameters:

System State Components:

1) "the request is not rejected":<bool>.
2) "the operator is not a library staff or borrower":<bool>.
3) "borrower's account doesn't exist":<bool>.
4) "borrower's account is not active":<bool>.
5) "borrower’s identifier doesn’t exist":<bool>.
6) "the borrower has to make a payment":<bool>.
7) "the item is not on a local charge":<bool>.
8) "the item is on Interlibrary charge":<bool>.
9) "internal department is not mending or binding":<bool>.
10)" the due date occurs when the borrower’s card is expired":<bool>.
11)" the number of renewals is more than a possible limit":<bool>.
12)" the due date occurs when the library is closed":<bool>.
13)" the item is overdue":<bool>.

Stimulus Response Components:

1) <renew item request>:<request message>.
2) <renew item rejection>:<response message>.
3) <acceptance>:<response message>.
4) <renew item response>:<response message>.
5) <renew loan period request>:<request message>.
6) <renew loan period rejection>:<response message>.
7) <renew loan period response>:<response message>.

Local Functions:
Appendix D. Library Example

Local Predicates:

1) "The response times for the I/O transactions described in this specification unit shall be in Class" (<integer>) is a predicate.

2) "The number of renewals has a default value of" (<integer>) "times" is a predicate.

External Functions:

External Predicates:

Exported Functions:

Exported Predicates:

Performance:

1) {The response times for the I/O transactions described in this specification unit shall be in Class {2}}.