Formalizing a Structured Natural Language Requirements Specification Notation

Kendra Cooper
University of Texas at Dallas
MS 31 P.O. Box 830688
Richardson, TX, USA 75083-0688
kcooper@utdallas.edu

Mabo Ito
University of British Columbia
2356 Main Mall
Vancouver, BC, Canada V6T 1Z4
mito@ece.ubc.ca

Abstract. Requirements specification notations are developed by organizations in order to meet their specific needs. For example, the Threads-Capabilities notation, an in house notation at Raytheon Systems Canada, Ltd., has been developed and used for specifying their complex, large scale, air traffic control systems. It is a semi-formal, structured, natural language notation. In this work, we investigate how to make this semi-formal notation more rigorous (i.e., formal) by developing and applying a new formalization process to it. By doing this, we can obtain the advantages of formal methods (precise, unambiguous, automatic generation of test specifications, automated typechecking, etc.) while retaining the style and readability of the original notation. We call the formalized notation the Stimulus Response Requirements Specification (SRRS) notation. Our results have been successful for the specific notation. The formalized notation has been demonstrated to reduce the time and improve the quality of the requirements specifications. There is additional training time, however, needed to learn to use the notation and tools.

INTRODUCTION

The idea of applying a process to a semi-formal notation and formalizing it is very appealing because it has the potential to offer a familiar notation that is precise, unambiguous, and is amenable to automated tool support. The new notation has the potential to overcome a commonly stated disadvantage of formal methods: they are difficult to read, write, and understand.

Recent examples of semi-formal notations that have been (partially) formalized include structured analysis (Larsen 1994, Leavens 1999) and the Unified Modeling Language (UML) (Ober 2000, McUmber 2001). In such work, the notations are clearly and precisely defined. The process, however, to go from the semi-formal to the formal is generally not described. If we needed to take a different semi-formal notation and formalize it, then where would we start?

To help answer this question, we have developed and applied a process to formalize an existing requirements specification notation. In this work, the process begins with a semi-formal notation called Threads-Capabilities that has been used at Raytheon Systems Canada, Limited (Paine 1993). The Threads-Capabilities notation has been used on the CAATS and MAATS projects to document the software requirements specifications. These complex, large scale projects have contracts valued at 500 million and 73 million Canadian dollars respectively (Auditor 1996, Raytheon 2001). The software requirements specification for the CAATS project is documented in approximately 3100 pages. The Threads-Capabilities 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 need 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.

As this is our first attempt to develop and apply a formalization process, we recognize that the proposed process needs to be used, evaluated, and refined by applying it to formalize a variety of different semi-formal notations.

This paper is organized as follows. Section 2 provides an overview of the existing Threads-Capabilities notation. The formalization process is presented in Section 3. Conclusions and future work are in Section 4.

THE EXISTING NOTATION

The Threads-Capabilities technique has been developed as an in house requirements specification notation for two air traffic control 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 that connects an external event or stimulus to an output event or response (Deutsch 1988). 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.

The Threads-Capabilities approach is a structured, natural language specification approach. A thread is composed of a title, overview, stimulus, response, requirements, and performance section.

The title provides a unique identifier for the thread in the specification document. The title is a short, meaningful description of the thread.

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.

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 that is referred to in the requirements section.

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.

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.

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.

The Threads-Capabilities notation defines writing style conventions that 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 automatically 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 Threads-Capabilities technique.

FORMALIZATION PROCESS

The process used to formalize the Threads-Capabilities notation, defined in IDEF0 (FIPS 1993), has five steps (refer to Figure 1). Each of the steps is described in the sections below.

Evaluate Threads-Capabilities Notation. In order to improve the notation, it is necessary to evaluate its current strengths and weaknesses. The Threads-Capabilities notation's main strengths include:

1. external partitioning of the requirements
2. blackbox style
3. abstraction, or grouping, of the stimuli and responses
4. readability of the notation
5. highly structured format
6. definition of terminology with a data dictionary

The Threads-Capabilities notation has a number of drawbacks, however. These disadvantages include:

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

Draft Semi-formal SRRS Notation. The second step in the process is used to create the semi-formal SRRS notation. The semi-formal SRRS notation is described in detail in (Cooper 1997); a sample specification unit from an on-line library system is in Appendix A.

A simple example of a requirement statement written in semi-formal SRRS for a library system is:

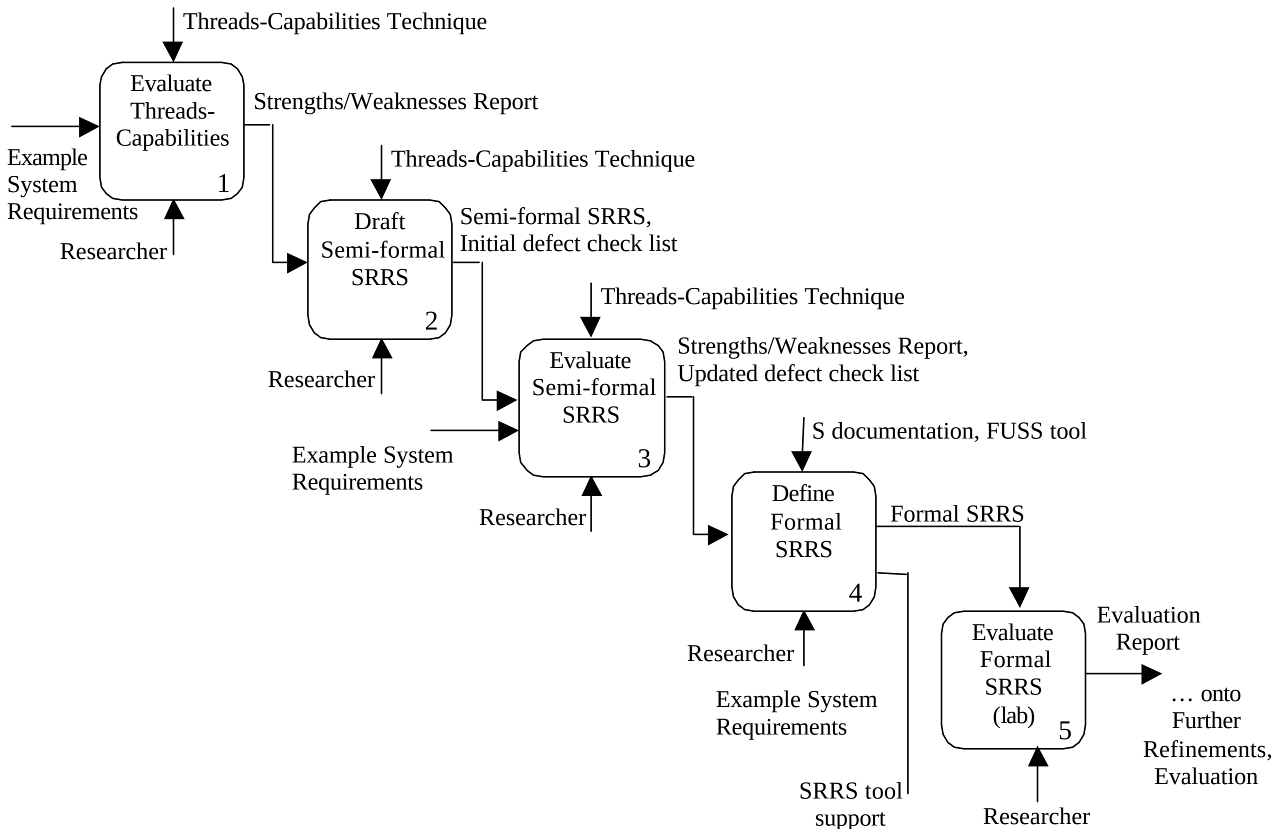
```
1. Upon receipt of a [sign out book
request], if all of the following
conditions are true:
   a) The borrower has privileges at
      the library
   b) The borrower does not have
      outstanding fines
   c) The book is allowed to be
      signed out
the system shall send a [charge book
response].
```

Two of the problems identified in the Threads-Capabilities 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. The second correction is to clearly define a re-use mechanism for the requirements. In large systems, a set of requirements 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 signaling messages is defined that allow the author to clearly identify the requirements being re-used. This mechanism is retained in the formal SRRS notation.

Evaluate Semi-formal SRRS Notation. In the third step of the formalization process, the SF SRRS notation is evaluated (Cooper 1997). 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. Typechecking and type inference errors are categorized together, as they are often intermingled in the static checking process. 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



Threads-Capabilities Technique
Evaluate Threads-Capabilities 1
Strengths/Weaknesses Report
Example System Requirements
Researcher
Threads-Capabilities Technique
Draft Semi-formal SRRS 2
Semi-formal SRRS, Initial defect check list
Researcher
Threads-Capabilities Technique
Evaluate Semi-formal SRRS 3
Strengths/Weaknesses Report, Updated defect check list
Example System Requirements
Researcher
S documentation, FUSS tool
Define Formal SRRS 4
Formal SRRS
Researcher
Example System Requirements
SRRS tool support
Evaluate Formal SRRS (lab) 5
Evaluation Report
… onto Further Refinements, Evaluation
Researcher

Figure 1. The Formalization Process

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.

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. 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 are developed in lex and YACC.

To define the semantics, the SRRS notation is translated into S, a higher order logic that used in our research 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:

1. 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;
   ```

2. 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;
   ```

   The constant declaration below is a predicate declaration:

   ```
   “the book has been charged out
   to” : borrower ->bool;
   ```

3. 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.

   ```
   :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
   ```

4. 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 a typechecking using the S tool support. The S notation has a number of rules that must be adhered to if the translation is to successfully typecheck. 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. The S tool can only typecheck the function applications that are explicitly provided.

The mapping from SRRS to S is summarized in Appendix B. After the translations are done manually, algorithms are developed that can be used to automate the procedure. 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
            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 1. The complete set of translations is available in (Cooper 2001).

SRRS Predicate Declaration	S Constant Declaration
1) "This is a flex-fix predicate declaration because the formal arguments (<type 1>) and (<type 2>) are inter-mingled with the body of the predicate" is a predicate.	"This is a flex-fix predicate declaration because the formal arguments <type 1> and <type 2> are intermingled with the body of the predicate" : <type 1> -> <type 2> -> <bool>;

Table 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.

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 experiment is based on (Porter 1995). 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):

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.

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.

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.

Where:
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.

The results of the experiment 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 by 39% and the number of defects detected in a peer review process is reduced by 81%. There is, however, an increase in the training time for the authors (Cooper 2000).

CONCLUSIONS AND FUTURE WORK

The formalization process used in this work is a systematic, step by step process that begins with a semi-formal notation, Threads-Capabilities, 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 (Donat 1998). 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.

As this is our first attempt to develop and apply a formalization process, we recognize the proposed process needs to be used to formalize a variety of semi-formal notations, evaluated, and refined. To obtain an accurate, quantitative understanding of the costs and benefits of formalizing a notation, the time to apply the process and to develop the tool support needs to be measured. In this preliminary work, these metrics were not collected.

REFERENCES

Cooper, K., Joyce, J., and M. Ito, "Advantages of Stimulus Response Requirements Specification Techniques", The University of British Columbia, CICSR technical report TR97-001, 1997.

Cooper, K. and Ito, M., "Experimental Evaluation of the Stimulus Response Requirement Specification Notation", April 17-19, Staffordshire, UK, EASE 2000.

Cooper, K., “Stimulus Response Requirements Specification Notation: An Empirically Evaluated Requirements Specification Notation", Ph.D. thesis, The University of British Columbia, 2001.

Deutsch, M. and Willis, R., *Software quality engineering: a total technical and management approach*, Prentice Hall Inc., USA, 1988.

Donat, M., Cooper, K., and Ito, M., "Capturing the logical structure of requirements for the automatic generation of test specifications", EKA '99, May 26-28, 1999, Braunschweig, Germany, pp 567-582.

Federal Information Processing Standard (FIPS) Publication 183, Integration Definition for Function Modeling (IDEF0), December 21, 1993.

Larsen, P., Fitzgerald, J., and Brooks, T., “Applying formal specification in industry”, IEEE Software, May 1996, pp. 48-56.

Larsen, P., Plat, N., and Toetenel, H., “A formal semantics of data flow diagrams”, Formal aspects of

Computing, 6(6), December 1994, pp. 586-606.

Leavens, Gary, Wahls, T., and Baker, A., "Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages", ACM Symposium on Applied Computing, 1999, pp. 526-532.

Levin, J., Mason T., and Brown, D., lex and yacc, O'Reilly & Associates Inc., USA,1992.

McUmber, W. and Cheng, B., "A general framework for formalizing UML with formal languages", Proceedings of the 23rd Int. Conf. on Software Engineering, 2001, pp. 433-442.

Ober, I., "More meaningful UML models", Proceedings of 37th Int. Conf. on Technology of Object-Oriented Languages and Systems, 2000, pp. 146-157.

Paine, T., Krutchen, P., and Toth, K., "Modernizing ATC Through Modern Software Methods", 38th Annual Air Traffic Control Association Convention, Nashville, Tennessee, October, 1993.

Porter, A., Votta, L., and Basili, V., "Comparing detection methods for software-requirement inspections: A replicated experiment", IEEE Transactions on Software Engineering, 21 (6), June 1995, pp 563-575.

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

Report of the Auditor General of Canada 1996, Chapter 24.

APPENDIX A. SAMPLE SPECIFICATION

A partial specification unit from an on-line library system is presented in the semi-formal SRRS notation.

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) Dallas Public Library <search request>
d) Vancouver Public Library <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) Dallas Public Library <search response>
d) Vancouver Public Library <search response>

Requirements:
1) Upon receipt of a [search request], if the input is not rejected, the system shall return an [acceptance].

2) Upon receipt of a [search request], the system 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) Dallas Public Library:<source>
4) Dallas Public Library:<destination>
5) Vancouver Public Library:<source>
6) Vancouver Public Library:<destination>

Local Predicates:
1) The response times for the I/O transactions described in this specification unit shall be in Class <integer>.

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

APPENDIX B. SRRS TO S MAPPING

SRRS Component	S Translation
Title, Overview, Stimuli, Responses	N/A
Performance	Predicate Application or Boolean expression
Logical Condition	Predicate Application or Boolean Expression
Data Expression	Function Application or Data Object
Type Name Declaration	Type Declaration
Subtype Declaration	Type Declaration (unique name guaranteed) Constant Declaration (function, unique name guaranteed) Constant Declaration (unnamed data object)
Static Association Declaration	Constant Declaration (unique name guaranteed)
Constant Declaration	Constant Declaration (unique

	name guaranteed)
Variable System Parameter Declaration	Constant Declaration (unique name guaranteed)
System State Component Declaration	Constant Declaration (unique name guaranteed)
Stimulus Response Component Declaration	Type Declaration (unique name guaranteed) Constant Declaration (function, unique name guaranteed) Constant Declaration (unnamed data object)
Local Function Declaration	Constant Declaration
Local Predicate Declaration	Constant Declaration
External Function Declaration	Constant Declaration
External Predicate Declaration	Constant Declaration
Exported Function Declaration	Constant Declaration
Exported Predicate Declaration	Constant Declaration
Stimulus condition response condition requirement	if (receive stimulus AND condition) then (response AND condition
Stimulus condition response requirement	if (receive stimulus AND condition) then (response)
Stimulus response requirement	if (receive stimulus) then (response)
Condition response condition requirement	if (condition) then (response AND condition)
Condition response requirement	if (condition) then (response)

BIOGRAPHY

Dr. Kendra Cooper is an Assistant Professor in the department of Computer Science at The University of Texas at Dallas. She received a Ph.D. in Electrical and Computer Engineering from The University of British Columbia. Her research interests include requirements engineering, formal methods, and the empirical assessment of software engineering methodologies.

Dr. Ito is a professor in the Department of Electrical and Computer Engineering at The University of British Columbia. His broad range of research interests include parallel processing, real-time systems, computer communications, and image and signal processing.