Assessment of a Lightweight Formal Method for Specifying and Analyzing Requirements

K. Cooper
The University of Texas at Dallas Erik Jonnson School of Engineering and Computer Science
kcooper@utdallas.edu

Tim Menzies
The University of British Columbia Department of Electrical and Computer Engineering
Tim@menzies.com

Mabo Ito
The University of British Columbia Department of Electrical and Computer Engineering
mito@ece.ubc.ca

Abstract

Tools in requirements engineering are recognized as a key component in achieving the goal of building systems better, faster, and cheaper. Tools don't get distracted, don't need a lunch break, don't have another meeting to attend, don't make (as many) mistakes, and they don't get bored. Despite this, few of the research tools in RE are being adopted in the broader SE community. Why?

Our thesis is that many of the research tools excel at tasks that are not interesting to commercial practitioners. For example, users are more likely to use a tool if it can be quickly applied to their current practice and many research tools are not widely applicable.

The article describes a general approach for the construction and evaluation of domain-specific RE tools of high applicability. By using this methodology, it can be shown that developing a notation and its tool support is cost-effective. The evaluation is so precise that a "break-even" point can be defined after which this approach is clearly useful.

1. Introduction

Tools in requirements engineering are recognized as a key component in achieving the goal of building systems better, faster, and cheaper. Tools don't get distracted, don't need a lunch break, don't have another meeting to attend, don't make (as many) mistakes, and they don't get bored. One of the puzzles in modern RE research is the increasing number of tools being developed without any hard evidence that these tools are widely accepted in the community of RE practitioners. This is not to say that, in isolated case studies, these tools have not proved useful for RE. Many such case studies exist based on using the SCR tool, the SPIN tool, and the KAOS tool [1,2,3]. However, looking beyond the particulars of these studies, what can we say to commercial RE practitioners in order to convince them to use these tools?

Our thesis is that we lack a convincing demonstration of the power of these tools for several reasons. The first reason is that these tools excel on tasks that are not interesting to commercial practitioners. For example, the SPIN tool optimizes for completeness and expressive power of the temporal logic queries [4]. The SP2 tool optimizes for the tractability of the inference procedure [5]. However, in our experience, commercial practitioners care less for these issues and more about how understandable the tools are.

A second issue, also related to understandability, is that users are more likely to use a tool if it is immediately applicable, i.e., it is compatible with how things are already being done. A tool that optimizes for applicability would not require a time consuming and confusing translation of the local domain documents into a format suitable for the RE tool.

Thirdly, we lack convincing experiments regarding the merits of a particular tool or the relative merits of different tools. In order to assess the results of such experiments, we need to clearly identify the goals of the tool. Hence, the above two points are vital for the design of convincing experiments.

In this article we report an experiment that evaluates two techniques for requirements capture and analysis. Two key parts of this evaluation are 1) The definition of a "break-even" point, below which the experimental tool is not useful and above which it is considered desirable. Secondly, our tools optimize for applicability; i.e., how well they do they support the already occurring documents in the domain. Note that, in our view, such an assessment experiment must be defined alongside the development of any new tools. An objection to the use of some new tool, particularly if that tool has domain specific elements, is that new tools can be ad hoc, time consuming to build, and hard to assess.

The article is structured as follows. Section 2 discusses related work. Section 3 presents the process we use to develop our formal notation and the results of our assessment of the notation. The conclusions and future work are presented in Section 4.

2. Related Work

Requirements engineering techniques have been assessed, or evaluated, using case studies, pilot projects, and experiments [7,8,9,10]. Evaluations are used to discover strengths and weaknesses in a technique, determine the costs of applying a 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 [11]. 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.

Much recent research explores the merits of various tools for assessing requirement models. In this related work section, we comment on three examples of that research: KAOS [2], applications of the SPIN tool to [3], and SCR [1]. In addition, we discuss natural language based techniques in terms of their strengths and weaknesses.

In the KAOS system, analysts generate a properties model by incrementally augmenting object-oriented scenario diagrams with temporal logic statements. The KAOS methodology is tightly linked to standard object oriented (OO) methods and extends them with temporal logic constraints. For domains that use OO, KAOS would score high on our applicability scale, i.e., KAOS would be a useful way to explore OO requirements documents. In domains where the locals use some specific, possibly idiosyncratic, extension to established methodologies, general tools like KAOS would have to be adapted in order to satisfy the applicability criteria. For example, in the domain studied here, the existing documents adopted the functional style of the Threads-Capabilities system. The tool discussed above was required in order to support the particulars of that tool. One argument against developing a local tool for a local dialect is that such a tool can be complex to develop and hard to assess. In the work done here, a novice to compiler theory could use off-the-shelf technology (LEX and YACC) to develop such a local tool. Further, that same developer could conduct a well-designed experiment to assess the utility of that tool versus some existing method.

In other work, Schneider [3] used SPIN, a full-featured temporal logic model checker, to assess NASA documents. Such full-featured model checkers can incur an extremely high start-up cost as analysts struggle to fit their knowledge into the syntax of that model checker. To reduce this cost to a manageable level, Schneider used lightweight formal modelling, i.e., only partial descriptions of the systems and properties models were constructed. Despite their incomplete nature, Schneider found that such partial models could still detect significant systems errors. While exciting research, this approach still scores low on our applicability scale since the naturally occurring models had to be contorted to be processed by SPIN.

The SCR notation and parts of the SCR toolset have recently been used in many applications including a recent case study for a light control system [1]. In this work, a requirements specification was developed for a system that controls the lighting in an office building. Three tools in the SCR toolset were used: the automated consistency checker; the simulator; and a tool (Salsa) that analyzes a specification for desired properties. These tools were been selected for use because applying these tools is relatively easy. Other tools in the toolset have been recognized as requiring more effort: the SPIN model checker and the TAME tool (an interface to the PVS theorem prover). The requirement specification contains many numbers and large ranges of numbers (e.g., a light level can range from 0 to 10,000), which lead to a large state space. The state space explosion makes this requirement specification less suitable for using a model checker. Using the PVS system requires more effort because additional properties must be defined for the light control system. Although using these additional tools would have provided more confidence in the requirements specification, the researchers chose not to use them because of their additional cost.

Use case and scenario based requirements specification techniques are popular choices in industry today. Both families of specification techniques use natural language, which make the requirements readable and understandable, and encourage a user-centric partitioning of the requirements, which make the requirements straightforward to review for completeness. However, if natural language techniques are compared to formal methods, then some significant drawbacks with the natural language based approaches become apparent. One significant difference is the level of tool support that is available for use.

Formal methods offer a wide range of tools that have automated two major tasks in software development: the analysis of requirements for defects and the generation of test specifications. The tools available for detecting defects include parsers, typecheckers, consistency checkers, simulators, model checkers, and theorem proving systems [1,4, 12,13,14,15]. These tools can be used to find and correct defects as early as possible in the software development lifecycle. 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 [16]. 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 result is an improvement in the quality of the requirements and a reduction in the time and cost to develop them.

Although numerous formal methods are available, they have not been applied routinely in industry. The lack of readability for methods such as Z, VDM, or the HOL meta-language by customers who are not trained in formal methods is a significant deterrent. In addition to the customer struggling with the notation, the requirements authors and the design team must learn to use the notation and the tool support effectively. The classroom and on the job training time to accomplish this is high. For example, an introductory classroom course in the Z notation is five days long.

The natural language approaches and the formal methods appear to be at extreme ends of a scale for requirements specification approaches. If the priority is on having readable requirements, then a natural language approach is the appropriate choice and automated tool support is sacrificed. If the priority is on having automated tool support, then a formal method is the appropriate choice, and readability is sacrificed. Ideally, a notation that is readable *and* provides tool support could be developed. The solution presented in this work is a lightweight formal method. The notation is designed to support the automated detection of grammar and typechecking defects. The notation also supports the automated generation of system level test specifications [22].

3. Developing a Formal Notation

Our process for developing the new notation has been organized in five basic steps (refer to Figure 1). It is important to note that one of the goals of formalizing the notation is to keep it as similar as possible to the one already used in industry. Accomplishing this goal is important in retaining the high applicability rating of the notation. The first step is to evaluate the strengths and weaknesses of the existing, semi-formal notation that has already been used in industry. The second step is to correct deficiencies found in the notation. For example, a problem discovered in the semi-formal notation is the lack of a definition of how to match the stimuli and the responses. The third step is to define the syntax and the semantics for the new notation. The fourth step is to evaluate the new notation in terms of its costs and benefits. The fifth step is to move the notation out of the lab and into a small, pilot study in industry. This step allows the notation and tool support to be exercised and evaluated in a real world setting. At this point, the notation is likely to need updates.

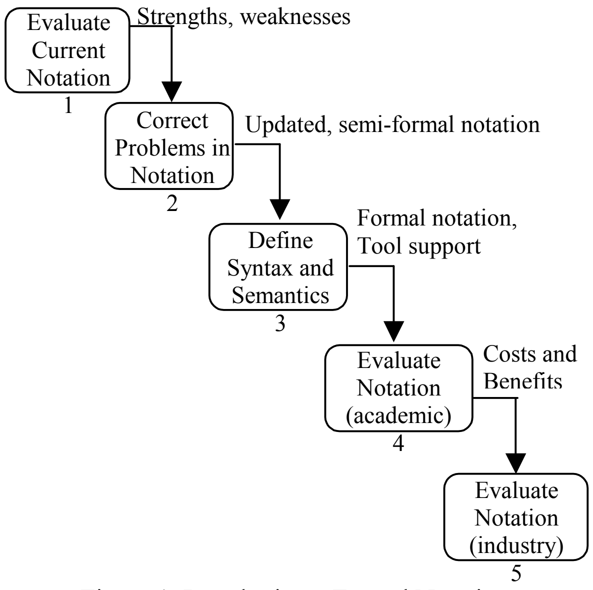
Evaluate Current Notation
1
Strengths, weaknesses
Correct Problems in Notation
2
Updated, semi-formal notation
Define Syntax and Semantics
3
Formal notation, Tool support
Evaluate Notation (academic)
4
Costs and Benefits
Evaluate Notation (industry)
5

Figure 1. Developing a Formal Notation

3.1 Evaluating The Existing Notation

The requirements specification technique presented in this paper is based on a technique that has been used at Raytheon Systems Canada Limited on two complex, large scale, air traffic control systems: the Canadian Automated Air Traffic Control System (CAATS) and the Military Automated Air Traffic Control System (MAATS). These projects have contracts valued at 500 Million and 73 million respectively [17,18]. The software requirements specification for the CAATS project is documented in approximately 3100 pages.

The in-house notation at Raytheon is called the Threads-Capabilities technique [19]. A thread is a specification unit that specifies the actions performed by the system as the result of receiving 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. 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.

There are a number of strengths for the Thread-Capabilities technique. The technique:

- is based on natural language. Using English as the notation, the requirements are considered readable and understandable. The training time is also low because the notation is already familiar to the trainees.
- prescribes a user-centric (i.e., external) partitioning of the requirements. Each thread or capability describes a task the user needs to perform. This organization of the requirements eases the development and validation of the requirements specification document because a reviewer can work through a listing of the titles of the threads and capabilities and look for missing, duplicated, or extraneous threads or capabilities. This organization of requirements specification documents is scaleable and maintainable. When a new task is identified, it can be added in with little impact on the document.
- provides template phrasing for the authors to re-use. The template phrasing accomplishes three important things. First, the template phrasing promotes consistency among a group of authors. Without recommended or standardized phrasing, each author is left to devise their own writing style. On projects with a large team of requirements authors, this could lead to an inconsistent document. The template phrasing contributes to the ease of writing, reviewing, and validating the requirements. The template phrasing is also designed to promote a black-box style. The verbs used in the template phrasing are restricted to a set of externally visible action verbs including, for example, "send" and "update". The black-box style reduces the inclusion of details that may need to be updated as the software development lifecycle progresses.
- has a mechanism in the notation that allows the author to generalize a set of stimuli or responses into a group. This is convenient for large systems because stimuli from different sources may all trigger the same requirement and responses for different destinations may all be the result of a single requirement. It is convenient to describe the requirement once, rather than repeating the requirement for each possible source and destination.
- has an integrated data dictionary. The data dictionary is a repository for the names and definitions of the data used in the system as well as documenting the relationships among the data elements. For large, complex systems, having a clear and consistent set of terms is critical in developing correct and consistent requirements.

Even with all of the strengths of the Threads-Capabilities technique, there are areas in which the technique could be improved. For example, improvements include:

- develop tool support to check for conformance to the recommended template phrasing. The Threads-Capabilities technique only has a manual, peer review process for detecting and subsequently correcting defects in the requirements. The manual process is time consuming and, as a result, expensive. To support this, the syntax of the notation must be defined.
- define how the stimuli (inputs) in a group are matched with responses (outputs) in a group. The technique does not explicitly define these rules. Without a clear definition of the behaviour of the notation, each author and reviewer is left to create their own set of rules. This can lead to inconsistent interpretation of the requirements and an incorrect implementation.
- develop tool support to automatically generate the system level test specifications. To support this, the syntax and semantics of the notation must be defined. Without these definitions, the generation of test specifications is a manual process, which is time consuming and prone to errors.
- develop training material for new users. A description of the technique is available, however, there is no tutorial style training material available.

3.2 Updating the Existing Notation

After the evaluation of the Threads-Capability notation is complete, the notation is updated to correct a significant problem. The notation is updated to explicitly define the matching rules for stimuli and responses. For example, if the following groups of stimuli and responses are defined:

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

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

Responses:

1) As specified by the requirements described below, the library system shall send a [search response] to:
   a) Operator <search response>;
   b) VPL <search response>;
   c) SFU <search response>;
   d) UVIC <search response>;
   e) EPL <search response>.

, then this requirement written by the author:

1. Upon receipt of a [search request], if the {borrower does not owe fines} then the library system shall return a [search response].

is actually representing the following five requirements:

1. Upon receipt of a Operator <research request>, if the {borrower does not owe fines} then the library system shall return a Operator <search response>.
2. Upon receipt of a VPL <research request>, if the {borrower does not owe fines} then the library system shall return a VPL <search response>.
3. Upon receipt of a SFU <research request>, if the {borrower does not owe fines} then the library system shall return a SFU <search response>.
4. Upon receipt of a UVIC <research request>, if the {borrower does not owe fines} then the library system shall return a UVIC <search response>.
5. Upon receipt of a EPL <research request>, if the {borrower does not owe fines} then the library system shall return a EPL <search response>.

At this point, the notation is called a semi-formal notation and is given the name semi-formal Stimulus Response Requirements Specification (SRRS) notation.

3.3 Defining the Syntax and Semantics

The next step in developing the new notation is formally defining the syntax and semantics. The syntax of the notation is defined in BNF. The semantics of the notation are operationally defined using a translation into higher order logic. To demonstrate the feasibility of the notation and to support running the experimental evaluation, a tool is also developed. The tool is implemented in approximately 19 KSLOC of lex, yacc, and c code. The new notation is called the formal Stimulus Response Requirements Specification (SRRS) technique.

3.4 Evaluating the Updated Notation

The formal SRRS technique is objectively evaluated using a well defined experiment. 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 are quantified in the experiment. 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 defined as three test hypotheses. The first test hypothesis is that 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. The second test hypothesis is that 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. The third test hypothesis is that 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.

3.4.1. Results. The experimental results including defect rates, training time, and the time to write, review, and correct the specification units are summarized in this section. Group 1 in the results refers to the control group using the semi-formal version of the requirements specification notation. Group 2 refers to the experimental group using the formal notation.

The experimental results for the defect rates are summarized in Table 1. The results show a reduction in the syntax, type, and the analysis defects detected for Group 2. The % difference between the Group 1 and Group 2 for the total number of defects detected per allocated requirement object identifier (ROID) shows an 81% reduction in detected defects.

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

Table 1. Summary of Defects Recorded

	Group 1	Group 2	% Difference
Number of syntax defects per ROID	0.99	0.09	-90.91
Number of type defects per ROID	0.74	0.01	-98.65
Number of analysis defects per ROID	0.88	0.39	-55.68
Number of total defects per ROID	2.61	0.49	-81.23

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 practice exercise. In addition, the amount of time it takes the in the experimental group to work through the tutorial for the SRRS tool is considered as formal training [21]. 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, because the informal training continues as the subjects write their specifications. 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 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 2. Summary of Training Time

	Group 1	Group 2	% Difference
Formal Training Time Minutes/author	420.00	835.00	98.81
Informal Training Time Minutes/ROID	0.32	5.02	1468.75
Total Training Time Minutes/author	448.33	1285	186.62

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 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 of 39%.

3.4.2. Discussion of the Results. 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.

Table 3. Summary of Writing and Reviewing Time

	Group 1	Group 2	% Difference
Average time to write per ROID in minutes	17.58	10.58	-39.82
Average time to review and correct per ROID in minutes	15.28	9.42	-38.35
Average total time per ROID in minutes	32.86	20.00	-39.44

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.

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 the following calculation can be used to estimate the training time: training time = A * T + R * D, where A is the number of authors, T is the training time per author for a given notation, R is the number of ROIDs and 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.

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 defects removed. As a result the specification units have a lower detected defect rate and have a higher quality. This reduces the overall project costs, since correcting defects in the software increases an order of magnitude for every phase the error is propagated. The sooner defects are discovered and corrected the better [20]. The goal is to correct the defects in the same phase they are introduced in. The most expensive errors to correct are those that are detected by the customer after delivery, and are traced back to being introduced in the software requirements phase. Multiple levels of code, design, and requirement products must be updated.

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.

With these experimental results an estimate of the development time for a project can be made. Given the notation proposed and number of allocated ROIDs to be written the following calculation can be used to estimate the development time with the simple calculation development time = R * D, where R is the number of ROIDs and 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 and the development time (refer to Table 4).

Table 4. Example of Total Time Estimates

	Semi formal notation	Formal notation
Training Time	20 * 420 + 2000*0.32 = 9042 minutes	20 * 835 + 2000 * 5.02 = 26740 minutes
Development Time	2000*32.86 = 65720 minutes	2000 * 20.00 = 40000 minutes
Total Time	9042 + 65720 = 74760 minutes	26740 + 40000 = 66740 minutes

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 slight advantage in selecting the formal notation. The caveat in the estimates made here is that the data used in the calculations is derived from one project with 432 allocated ROIDs. The data has not been confirmed in different projects of different sizes.

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.

In industrial settings, these savings are significant. Engineers spend less time in peer review sessions recording minor defects, such as formatting or grammar errors. Instead, they can focus on detecting the more significant problems, such as detecting logical errors of omission or commission in the requirements.

The new lightweight technique has limitations, however. For example, the defects that are currently detected are limited in the scope to a single specification unit. In the future, analyzing a set of specification units making up a requirements specification document is going to be necessary. A second limitation is the limited amount of analysis that is performed at the moment. Simulation, Model checking and theorem proving techniques have not been investigated with this notation, yet. The idea of developing an integrated suite of tools, like the one that supports the SCR notation, is very appealing. It would provide RE practitioners with options. The RE could select the level of checking that is suitable for their system.

4. Conclusions and Future Work

We have presented a general process for developing a formal, domain specific notation with tool support using off the shelf technology. The notation being developed using this process rates high on applicability because the new, formal notation retains the strengths of the original notation (looks like natural language, external partitioning, integrated data model), and overcomes the identified weaknesses (lack of definition of the matching mechanism for stimuli and responses, tool support, and training material).

A critical step in our process is the assessment of the technique. If a technique can be demonstrated to be cost-effective to use, then we can use this to convince RE practitioners that the technique is practical (at least in an academic setting) and that a pilot study would be extremely beneficial.

The benefits of using this notation with its tool support have been quantified in an experiment. The results of the experiment indicate there is an 81% reduction in the defects detected in peer reviews and a 39% reduction in the amount of time needed to write, review, and correct specifications. The costs of using the technique, however, are an increase in training time. In comparison with the updated Threads-Capabilities notation, the training time increased from one day to two days. However, this is still less than an introductory course in Z, which takes five days. From these results, we can estimate the break-even point (i.e., when the technique becomes cost effective to use). If we have as few as two authors writing up 200 requirements, the formal SRRS notation with its tool support is cost effective to use.

The next step in developing this new notation is to evaluate it in a pilot study in industry. In this step, the notation can be evaluated in terms of what is used in the notation (and should be kept), what is not used in the notation (and should be removed), and what is missing from the notation (and should be added).

An interesting branch for this research is to determine how to tailor this notation for use in other domains. Since the notation has been developed for stimulus response systems, it is not well suited for describing declarative requirements of the form "The system shall have 6 nines availability". Currently, this requirement would have to be awkwardly stated as follows: "if {TRUE}, then the system shall commit the 6 nines availability" in SRRS.

5. References

[1] C. Heitmeyer and R. Bharadwaj "Applying the SCR Requirements Method to the Light Control Case Study", Journal of Universal Computer Science (JUCS), August 2000.

[2] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", IEEE Transactions on Software Engineering, Special Issue on Scenario Management, Novermber, 1998.

[3] F. Schneider, S.M. Easterbrook and J.R. Callahanand , G.J. Holzmann , W.K. Reinholtz and A. Ko and M. Shahabuddin, "Validating Requirements for Fault Tolerant Systems using Model Checking", 3rd IEEE International Conference On Requirements Engineering, 1998.

[4] G. Holzmann, "The model checker SPIN", IEEE Transactions on Software Engineering, 23(5), pp. 279-295, May 1997.

[5] T. Menzies and J. Powell and M. Houle, Fast Formal Analysis of Requirements via 'Topoi Diagrams' *ICSE 2001*, Available from http://tim.menzies.com/pdf/00fastre.pdf, 2001.

[6] J. Britt, "Case study: Applying formal methods to the traffic alert and collision avoidance system (TCAS) II", Computer Assurance: COMPASS '94, pp 39-51.

[7] D. Craigen, S. Gerhart, T. Ralston, "Case study: Darlington nuclear generating station", IEEE Software, January 1994, Volume 11, pp 30-32.

[8] P. Larsen, J. Fitzgerald, and T. Brooks, "Applying formal specification in industry", IEEE Software, May 1996, pp 48-56.

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

[10] A. Porter and L. Votta, "Comparing detection methods for software-requirement inspections: A replication using professional subjects", Journal of Software Engineering, Volume 3, Number 4, December 1998, pp. 355-379.

[11] S. Bear, "Managing the introduction of formal methods", IEE Colloquium, 1991, No. 131: Managing critical software projects.

[12] J. M. Spivey, The FUZZ Manual, Computer Science Consultancy, UK, 1992.

[13] C. Heitmeyer, J. Kirby, B. Labaw and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements," Proc. Computer-Aided Verification, 10th Ann. Conf. (CAV'98), Vancouver, Canada, 1998.

[14] Gordon, M and Melham, T., Introduction to HOL A theorem proving environment for higher order logic, Cambridge University Press, 1993.

[15] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert, PVS System Guide version 2.3, SRI International Computer Science Laboratory, USA, 1999.

[16] A. Davis, "Identifying and Measuring Quality in a Software Requirements Specification", Software Metrics 1993 Symposium, pp 141 - 152.

[17] Report of the Auditor General of Canada 1996, Chapter 24.

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

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

[20] W. Humphrey, A Disciplined Approach to Software Engineering, Addison-Wesley Publishing Company, Inc., Canada, 1995.

[21] K. Cooper and M. Ito, Training Material and User Documentation for the Stimulus Response Requirements Specification Notation, CICSR Technical Report TR99-001, The University of British Columbia, 1999.

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