Using SCR Requirements

Marsha Chechik

University of Toronto
Department of Computer Science

October 5, 1998

Software Cost Reduction project originated over 20 years ago as an attempt to give a complete methodology for mathematically-precise development of software. Most of the successes of this project came from development and analysis of specifications. The SCR community was able to develop an easy to write and review requirements notation and a variety of techniques facilitating reasoning about the requirements and showed their effectiveness on a number of case studies. Thus, I find it important to document the following developments in this area:

- informal description and formal semantics of SCR;
- techniques to reason about correctness of SCR specifications;
- tool support for specifying and reasoning about systems in the SCR notation;
- applications of methods and tools to real-life projects (both successful and unsuccessful).

Developments in the SCR are centered around a group of researchers at NRL lead by Connie Heitmeyer. This group is doing a great job conducting research in the area of requirements analysis, and publicizing the SCR method among academic and industrial software engineers. The requirements notation is easy to read and review, has simple semantics, and has in general been liked by engineers who got to use it. There are industrial-quality automated tools, like SCR* [2], and a number of researchers outside NRL have used the method successfully on their projects. However, despite a growing number of successful case studies, it seems that the industry is slow adopting and embracing SCR as their development methodology. What is wrong? The following is a (partial) list of possible reasons:

- Industry perceives that the cost of constructing formal requirements, and SCR requirements in particular, can be prohibitively large, thus limiting a possible domain of application of the method to selected companies that develop safety-critical systems, e.g. Ontario Hydro and Rockwell Collins.

- Techniques for using SCR for creating designs of software are still sketchy. Notably, Parnas and his colleagues succeeded in giving the community a good idea on what the various parts of documentation associated with software should include and how they should relate to each other [4, 3]. However, their style of documentation might prove to be too rigorous for projects outside the safety-critical domain.

- There is little work on using SCR specifications in other phases of software development life cycle.

- Support for hierarchical development of SCR requirements is limited. Although CoRE [1] attempts to enable hierarchical, object-like construction of requirements, it is lacking formal semantics and tool support and therefore is not fully usable at this time.

- It is not very well understood how suitable the SCR notation is for projects which are subject to changes of requirements. After all, the methodology was developed with the Waterfall development cycle in mind, i.e., under the assumption that the requirements phase is completed before all other phases begin. Unfortunately, this is a problem with virtually all formal methods.

In our opinion, the first and the third items of the above list are related. After all, being able to reuse SCR specifications in other development and analysis activities (preferably combined under a common interface), might make precise software construction more feasible from the financial point of view. These specifications can be used for automatic generation of test cases for black-box testing of code, for reasoning about code correctness, etc. Before starting to develop tools, let's imagine how they can be used in combination. The following is a sample methodology for using SCR throughout the software lifecycle, with support of a suite of tools under a common interface, called, for example, $SC(R)^3$ to stand for SCR Requirements Reuse:

1. A requirements designer provides an SCR tabular specification of the system's required behavior and optionally a set of desired properties of the system.

2. The specification is analyzed for conformance with these properties and simulated to give the designer an extra level of confidence that the specification is adequate. The analysis does not have to be fully automated, but if input from the user is required, it should be in a very intuitive form.

3. Once the specification is considered correct, a relationship between controlled and monitored variables and input and output devices (IN and OUT [3]) is specified. Design decisions, e.g. which processes are run in parallel and how the queues are organized, are specified in some extended SCR notation.

4. $SC(R)^3$ assists the user in generating an implementation. Based on the specification and the design decisions, it automatically generates a correct implementation for mode logic and some interface code. It also gives stubs for the remaining interface functions, living it to the user to fill in the details, e.g., for inputs from GUIs.

5. Optionally, the code is verified to ensure that assumptions of the mode logic part (NAT) correspond to the outputs of the user-provided part, i.e., the user-provided part is "correct".

6. Using the specification, $SC(R)^3$ generates test cases and test oracles [5]. Although the mode logic is correct by construction, the interface is likely to need testing.

7. The resulting implementation is analyzed for performance and optionally manually fine-tuned. If manual fine-tuning was used, the appropriate parts of the code are automatically checked for consistency with its specifications.

8. Once the problems pointed out by the analysis are fixed, the code is tested using (a subset of) test cases generated in step (6) above.

During the past several years, we have been working on some of the tools which will constitute a possible $SC(R)^3$ integrating them under a common interface. At this point, our $SC(R)^3$ allows users to specify and verify their requirements by invoking the SCR* toolkit, developed at the Naval Research Lab [2], and then uses these requirements to analyze correctness of annotated code. We also have a mostly operational prototype of a tool to generate black-box test cases. Finally, we are currently working on ways to use a hierarchical requirements notation like CoRE to collect necessary design information for code generation. Some preliminary results are available, although much more work is needed, and we invite other researchers to join this project.

In conclusion, we feel that adequate progress has been made in using SCR for requirements analysis, and hope to hear more about using these requirements for other analysis activities. It is important to survey what has been done in this area, both by our group and by other researchers, and join forces to develop tools and techniques to make SCR the most usable formal methodology. It deserves it.

References

[1] Stuart Faulk. "Software Requirements: A Tutorial". Technical Report NRL/MR/5546-95-7775, Naval Research Lab, Washington, D.C., November 1995.

[2] C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated Consistency Checking of Requirements Specifications". *Transactions on Software Engineering and Methodology*, 1996.

[3] David Lorge Parnas and Jan Madey. "Functional Documents for Computer Systems". *Science of Computer Programming*, 25:41–61, 1995.

[4] D.L. Parnas, J. Madey, and M. Iglewski. "Precise Documentation of Well-Structured Programs". *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

[5] Dennis K. Peters and David Lorge Parnas. "Using Test Oracles Generated from Program Documentation". *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.