SCR: A PRACTICAL METHOD FOR REQUIREMENTS SPECIFICATION

Constance Heitmeyer, Naval Research Laboratory, Washington, DC

Abstract

A controversial issue in the formal methods research community is the degree to which mathematical sophistication and theorem proving skills should be needed to apply a formal method. A premise of this paper is that formal methods research has produced several techniques with potential utility in practical software development, but that mathematical sophistication and theorem proving skills should not be prerequisites for using these techniques. In the paper, several attributes needed to make a formal method useful in practice are described. These attributes include user-friendly notation, automated (i.e., push-button) analysis, and easy to understand feedback. To illustrate the attributes of a practical formal method, a formal method for requirements specification called SCR (Software Cost Reduction) is introduced.

Formal Methods in Practice: Current Status

During the last decade, researchers have proposed numerous formal methods for developing computer systems. These include formal specification languages and formal analysis techniques, such as model checkers and mechanical theorem provers. One area in which formal methods have already had a major impact is hardware design. Not only are companies such as Intel beginning to use model checking, along with simulation, as a standard technique for detecting design errors, in addition, some companies are developing their own in-house model checkers. Moreover, a number of model checkers customized for hardware design have become available commercially.

In contrast, the use of formal methods in practical software development is rare. A significant barrier is the widespread perception among software developers that formal notations and formal analysis techniques are difficult to understand and apply. Many software developers also express serious doubts about the scalability and cost-effectiveness of formal methods.

An additional reason for the minimal impact of formal methods in software development is the absence in most software development processes of two features common in hardware design. First, hardware designers routinely use one of a small group of languages, e.g., Verilog or VHDL, to specify their designs. In contrast, precise specification and design languages are rarely used in software development.

Second, at hardware companies, integrating a formal method, such as a model checker, into the design process is relatively easy because other tools, such as simulators and code synthesis tools, are already a standard part of the design process. In software development, in contrast, no standard engineering process exists. For example, although “object-oriented design” (OOD) has become quite popular in recent years, OOD is still largely informal. Moreover, despite the recent availability of commercial CASE (Computer-Aided Software Engineering) tools, software developers rarely use CASE tools during the early stages of software development when such tools (and formal methods) have the greatest benefits.

While formal methods have significant potential for reducing software development costs and increasing software quality, the above barriers must be overcome before these benefits can be realized. Below, I recommend several improvements needed before formal methods can be useful in practical software development. I then introduce the SCR requirements method, an example of how formal methods can be used in practice.

Toward Practical Formal Methods

Described below are four general areas in which improvements in formal methods are needed. While most can be achieved by better engineering, in some cases, additional research is needed.

Minimize Effort and Expertise Needed to Apply the Method

To be useful in practice, formal methods must be convenient and easy to use. Currently, most software developers are reluctant to use formal methods because they

find the learning curve too steep and the effort required to apply the method too great. In many cases, deciding not to use a formal method is rational. The time and effort required to learn about and apply a formal method may not be worth the information and insight provided by the method; in some cases, the effort could be better spent applying another method, such as simulation. Suggested below are three ways in which the difficulty of learning and applying a formal method can be reduced.

- Offer a language that software developers find "natural". To the extent feasible, a language syntax and semantics familiar to the software practitioner should be supported. The language should have an explicitly defined formal semantics and should scale. Specifications in the language should be translated automatically into the language of a formal analysis tool, such as a model checker.
- Make formal analysis as automatic as possible. To the extent feasible, analysis should be "push-button", that is, the user should be able to invoke an analysis technique with the mere push of a button.
- Provide good feedback. When formal analysis exposes an error, the user should be provided with easy-to-understand feedback useful in correcting the error.

Provide a Suite of Analysis Tools

Because different tools detect different classes of errors, users should have available a "complete" suite of tools, carefully integrated to work together. Many hardware designers are already using suites of tools, such as simulators, model checkers, equivalence checkers, and code synthesis tools. One benefit of a suite of tools is that properties shown to hold using one tool may simplify the analysis performed by a second tool. For example, demonstrating that the specified system behavior is deterministic can simplify later analysis of the behavior with a model checker.

Integrate the Method into the User's Software Development Process

To the extent feasible, formal methods should be integrated into the existing user design process. Techniques for exploiting formal methods in object-oriented software design and in software development processes which use semiformal languages, such as Statecharts, should also be explored. How formal methods can be integrated into the user's design process should be described explicitly.

Provide a Powerful, Customizable Simulation Capability

Many researchers underestimate the value of simulation in exposing defects in software specifications. Unlike formal techniques which check the specification for properties of interest, simulation helps the user validate the specification. By using the simulator to symbolically execute the system, the user can ensure that the specified behavior captures his intent. One promising approach to selling formal methods is to build customized simulator front-ends, tailored to particular application domains. Such customized simulators can serve as system "prototypes", useful in demonstrating and analyzing the required system behavior prior to coding.

The SCR Requirements Method

The SCR (Software Cost Reduction) requirements method is a formal method based on tables for specifying the requirements of safety-critical software systems. Originally formulated in 1978 by NRL (Naval Research Laboratory) researchers to document the Operational Flight Program (OFP) requirements of the US Navy's A-7 aircraft, SCR has been used in practice by numerous organizations, including Grumann, Ontario Hydro, Bell Laboratories, and Lockheed, to specify software requirements.

The SCR method uses a tabular notation to specify requirements. Underlying the tabular notation is a state machine semantics. Specifications based on tables are relatively easy for software practitioners to understand and to produce. In addition, tables provide a precise, unambiguous basis for communication among practitioners and a natural organization for independent construction, review, modification, and analysis of parts of a large specification. Finally, tabular notations scale. Evidence of the scalability of tabular specifications has been demonstrated by engineers at Lockheed, who used SCR-style tables to specify the complete requirements of

the C-130J OFP, a program containing over 230K lines of Ada.

The SCR Toolset

Introduced in 1995, SCR* is an integrated suite of tools supporting the SCR requirements method. The toolset includes a specification editor for creating a requirements specification, a dependency graph browser for displaying the variable dependencies in the specification, a consistency checker for detecting errors such as type errors and missing cases, a simulator for validating the specification, and a model checker for checking application properties. Currently, more than 50 commercial, academic, and government institutions in the US, Canada, UK, and Germany, are experimenting with SCR*.

Specification Editor

To create, modify, or display a requirements specification, the user invokes the specification editor. Each SCR specification is organized into dictionaries and tables. The dictionaries define the static information in the specification, such as the user-defined types and the names and values of variables and constants. The tables specify how the variables change in response to input events. One important class of tables specifies the values of the system outputs.

Dependency Graph Browser

Understanding the relationship between different parts of a large specification can be difficult. To address this problem, the Dependency Graph Browser represents the dependencies among the variables in a given SCR specification as a directed graph. By examining this graph, a user can detect errors, such as undefined variables and circular definitions. The user can also use the DGB to display and extract parts of the dependency graph, e.g., the subgraph containing all variables upon which a selected controlled variable depends.

Consistency Checker

The consistency checker detects syntax and type errors, variable name discrepancies, missing cases, unwanted nondeterminism, and circular definitions. When an error is detected, the consistency checker provides detailed feedback to facilitate error correction by displaying the table (or dictionary) containing the error and highlighting the erroneous entries. It also provides a “counterexample”, i.e., an example that demonstrates the error. A form of static analysis, consistency checking is usually less expensive computationally than model checking. In developing an SCR specification, the user normally invokes the consistency checker first and postpones more expensive analysis, such as model checking, until later. Exploiting the special properties guaranteed by consistency checking (e.g., determinism) can make later analyses more efficient.

Simulator

To validate a specification, the user can run the simulator and analyze the results to ensure that the specification captures the intended behavior. Additionally, the user can define properties believed to be true of the required behavior and, using simulation, execute a series of scenarios to determine if any violate the properties.

The simulator supports the construction of front-ends, tailored to particular application domains. One example is a customized front-end for pilots to use in evaluating an attack aircraft specification (see Figure 1). Rather than clicking on variable names, entering values for them, and seeing the results of simulation presented as variable values, a pilot clicks on visual representations of cockpit controls and views the results on a simulated cockpit display. This front-end allows the pilot to move out of the world of requirements specification and into the world of attack aircraft, where he is the expert. Such an interface facilitates validation of the specification.

Model Checker

Recently, the explicit state model checker Spin was integrated into SCR*. After using the SCR tools to develop a requirements specification, a developer can automatically translate the specification into Promela, the language of Spin, and then invoke Spin within the toolset to check that the specification satisfies properties of interest. The user can use the simulator to demonstrate and validate any property violation detected by Spin.

The number of reachable states in a state machine model of real-world software is usually very large, sometimes infinite. To make model checking practical, three push-button

techniques have been developed which derive sound abstractions from SCR specifications. The methods are practical: none requires ingenuity on the user's part, and each derives a smaller, more abstract specification automatically. For example, prior to invoking Spin to check a weapons system specification for a safety property, we used our abstraction methods to automatically reduce the number of variables from over 250 to 55 and to replace several real-valued variables with finite-valued variables. Without these reductions in the number of variables and the type sets of the real-valued variables, model checking would be infeasible.

Practical Use of the SCR Tools

To date, the SCR tools have been applied in three pilot projects external to NRL. In the first, researchers at NASA's IV &V Facility used SCR* to detect missing cases and instances of nondeterminism in the prose requirements specification of software for the International Space Station. In the second project, engineers at Rockwell-Collins used the tools to expose 24 errors, many of them serious, in the requirements specification of an example flight guidance system. Of the detected errors, a third were uncovered by entering the specification into the toolset, a third in running the consistency checker, and the remaining third in executing the specification with the simulator. In a third project, researchers at the JPL (Jet Propulsion Laboratory) used SCR* to analyze specifications of two components of NASA's Deep Space-1 spacecraft for errors.

In a fourth pilot project, NRL applied the SCR tools to a sizable contractor-produced requirements specification of the Weapons Control Panel (WCP) for a safety-critical US military system. The tools uncovered numerous errors in the contractor specification, including a safety violation which could result in the malfunction of a weapon. Translating the contractor specification into the SCR tabular notation, using SCR* to detect specification errors, and building a working prototype of the WCP required only one person-month, thus demonstrating the utility and cost-effectiveness of the SCR method.

Conclusions

The SCR tools can be distinguished in three major ways from commercial tools and other research tools. First, unlike most commercial tools for requirements specification, SCR* has a solid mathematical foundation, thus allowing mathematically sound analyses, such as consistency checking and model checking, unsupported by current CASE tools. Second, the SCR tools, unlike most research tools, have a well-designed user interface, are integrated to work together, and provide detailed feedback when errors are detected to facilitate their correction. Finally, users of SCR* can do considerable analysis without interaction with application experts or formal methods researchers, thereby providing formal methods usage at low cost.

Figure 1. Customized Simulator Front-End for an Aircraft Specification