Formal Methods for Specifying, Validating, and Verifying Requirements

Constance L. Heitmeyer
Naval Research Laboratory, Washington, DC 20375
email: heitmeyer@itd.nrl.navy.mil

I. Introduction

During the past three decades, many formal methods have been proposed whose purpose is to reduce the cost of constructing computer systems and to improve their quality. Informally, a *formal method* is a mathematically-based technique or tool useful in developing either hardware or software. Recently, formal methods have played a significantly increased role in hardware design. More and more companies that sell microprocessors and hardware chips, such as Intel, IBM, and Motorola, are using formally-based tools such as model checkers [1] and theorem provers (see, e.g., [2]), to detect flaws in their designs.

While formal methods are applied less frequently in software development, there have been a few recent cases in which they have detected previously unknown defects in real-world software. One prominent example is the result of research in Microsoft's SLAM project in which Ball and Rajamani designed several formal techniques to automatically detect flaws in device drivers [3]. In 2006, Microsoft released the Static Driver Verifier (SDV) [4] as part of Windows Vista, the latest Microsoft operating system — SDV uses the SLAM software-model-checking engine to detect cases in which device drivers linked to Vista violate one of a set of interface rules. Thus SDV helps uncover defects in device drivers, a primary source of software bugs in Microsoft applications.

One critical step in developing high-quality software is understanding and documenting the software requirements. Studies have shown that many software defects can be traced to ambiguous, incomplete, and inconsistent requirements specifications and that fixing these defects can be very costly, especially when the defects are detected late in development. A promising approach to constructing precise, complete, and consistent requirements specifications is to represent the requirements in a formal specification language and to check the specification for properties, such as completeness and consistency, with formal analysis techniques.

This article describes a requirements method called Software Cost Reduction [5], [6], [7], originally developed by Parnas, Heninger, and other researchers at the Naval Research Laboratory (NRL) starting in the late 1970s. A major NRL research goal was to evaluate the utility and scalability of software engineering principles by using the principles to reconstruct software for a practical system. The SCR method was formulated and demonstrated by constructing a requirements specification [5] and several design documents [8] for the flight program of the U.S. Navy's A-7 aircraft.

SCR uses a tabular notation to represent the required behavior of a software system to make the requirements *understandable* to practitioners. Once an SCR requirements specification has been formulated, a set of formally based tools called the SCR Toolset [9], [10] may be used to check the consistency, completeness, and correctness of the specification.

Section II reviews the semantics that underly SCR, introduces the SCR tabular notation, and provides an example of software requirements represented in the notation. Section III describes how the SCR tools may be used to check the consistency and completeness of requirements specifications, to validate that the specification captures the intended behavior, and to verify (i.e., prove), or refute, that the specification satisfies critical application properties. It also describes two recent tools: the SCR Test Case Generator and the SCR Code Generator.

II. SCR Formal Model and Tabular Notation

SCR Formal Model

The objective of an SCR requirements specification is to capture the required externally visible behavior of a software system precisely and unambiguously. In an SCR specification [6], [10], *monitored* and *controlled variables* represent, respectively, the quantities in the system environment that the system monitors and controls. The required system behavior is specified as relations the system must maintain between the monitored and controlled variables. To specify these relations concisely, the SCR language provides two types of auxiliary variables —*mode classes* and *terms*— as well as conditions and events.

A *condition* is a predicate defined on a system state. A basic *event*, represented as @T(c), indicates that condition c changes from false to true. The event @F(c) is defined by @T(¬ c). If c's value in the current state is denoted c and its value in the next state as c', then the semantics of @T(c) is defined by $\neg c \wedge c'$ and of @F(c) by $c \wedge \neg c'$. A *conditioned event*, denoted @T(c) WHEN d, adds a qualifying condition d to an event and has the semantics $\neg c \wedge c' \wedge d$.

In SCR specifications, the monitored variables are *independent variables*, and the mode classes, terms and controlled variables are *dependent variables*. SCR specifications define the values of the dependent variables using three types of tables: *condition*, *event* and *mode transition tables*. Each term and controlled variable is defined by either a condition or an event table. Typically, a condition table defines the variable values in terms of a mode class and a set of conditions; an event table defines variable values in terms of a mode class and a set of conditioned events. A mode transition table associates each source mode and a conditioned event with a destination mode. If the given event occurs in the source mode, then in the next state the system enters the destination mode

Two relations, NAT and REQ [11], define the relationship between the current and next state values of all monitored and dependent variables. NAT specifies the natural constraints on monitored and controlled variables, such as constraints imposed by physical laws and the environment. REQ uses the SCR tables to specify the required system behavior as constraints on the dependent variables. Given a set of dependent variables, REQ is defined as the conjunction of the functions defined by the tables. In SCR, a *state* is a function mapping a state variable name to a type-correct value; the required system behavior is defined as a state machine $\Sigma = (S, \theta, \rho)$, where S is the set of states, θ is a predicate on S which defines the set of initial states, and $\rho \subseteq S \times S$ is the transition relation which defines the allowable state transitions.

SCR Requirements Specification: Example

To illustrate the SCR notation, this section presents an SCR specification of a simple control system called the Safety Injection System (SIS) [12], which controls the water pressure level of a nuclear power plant's cooling system. The SIS specification indicates how the SIS responds to changes in its monitored variables by changing a single controlled variable, which controls whether safety injection is on or off. The specification uses a mode class (a set of modes) to capture the history of changes in the monitored variables. Although the SIS specification has only one controlled variable, most SCR specifications have many controlled variables. In SIS, the system starts safety injection (if it is not overridden) when water pressure drops below a certain constant value Low. In the SCR specification of SIS, the monitored variables —mBlock, mReset, and mWaterPres— denote the states of two switches, the block and reset switches, and the water pressure reading; the mode class mcPressure indicates one of three system modes, TooLow, Permitted, and High; the Boolean term tOverridden indicates whether safety injection is overridden; and the controlled variable cSafetyInjection indicates whether safety injection is on or off.

Fig. 1 shows the relationship between the SIS monitored variables, the SIS modes, and the single SIS controlled variable. When, for example, the system is in the mode TooLow and the water pressure reading changes from Low to greater than or equal to Low, the SIS mode changes to Permitted; similarly, when SIS is in the mode Permitted and the water pressure reading changes from a value greater than or equal to Low to less than Low, the SIS mode changes to TooLow.

Tables I–III contain the mode transition, event, and condition tables defining the transition relation for the SIS. Table I contains a mode transition table which defines the mode transitions for the mode class mcPressure. The first row of Table I states that if the system is in mode TooLow when water pressure changes from less than Low to a value exceeding or equal to Low, the system mode changes to Permitted. Table II contains an event table defining the term variable tOverridden. The entry 'Never' in the event table means that if the system is in the mode High, no event can cause tOverridden to change to true. The middle entry in the second row of Table II states that if the system is in either TooLow or Permitted and the user turns the Block switch on when the Reset switch is off, then the value of tOverridden in the next state is true. Table III is a condition table defining the value of a controlled variable cSafetyInjection as a state invariant. This invariant specifies the required relation between the values of cSafetyInjection, tOverridden, and mcPressure.

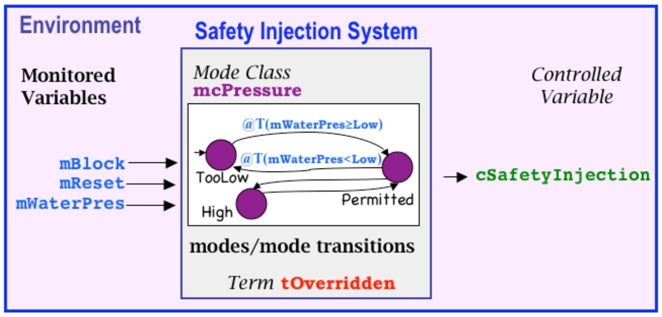

Fig. 1. SCR requirements specification of the Safety Injection System.

TABLE I
MODE TRANSITION TABLE FOR mcPressure.

Old Mode	Event	New Mode
TooLow	@T(mWaterPres $\geq$ Low)	Permitted
Permitted	@T(mWaterPres $\geq$ Permit)	High
Permitted	@T(mWaterPres $<$ Low)	TooLow
High	@T(mWaterPres $<$ Permit)	Permitted

TABLE II
EVENT TABLE FOR Overridden.

Mode Class mcPressure	Events	
High	Never	@F(mcPressure=High)
TooLow, Permitted	@T(mBlock=On) WHEN mReset=Off	@T(mcPressure=High) OR @T(mReset=On)
tOverridden$'$	True	False

TABLE III
CONDITION TABLE FOR cSafetyInjection

Mode mcPressure	Condition	
High, Permitted	True	False
TooLow	tOverridden	NOT tOverridden
cSafetyInjection	Off	On

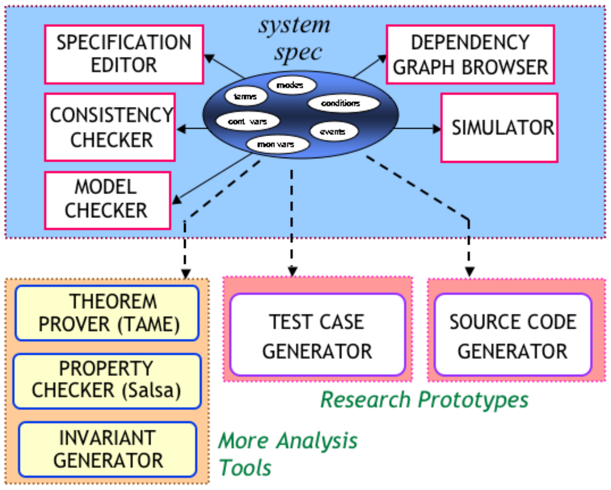

Fig. 2. SCR Toolset.

III. Tools

The SCR Toolset is an integrated suite of tools supporting the SCR requirements method [9], [10]. Fig. 2 illustrates the tools, which include a *specification editor* for creating and modifying a requirements specification, a *consistency checker* for checking the specification for well-formedness (e.g., type correctness), a *simulator* for symbolically executing the system based on the specification, a *model checker* for analyzing the specification for application properties, and a *dependency graph browser* for displaying variable dependencies. In addition, the toolset includes the TAME front-end to PVS, an invariant generator, a property checker Salsa, a test case generator, and a source code generator.

The SCR toolset has also been evaluated in numerous pilot projects. In one project, researchers at NASA's IV&V Facility used the tools to detect ambiguity and missing assumptions in a software requirements specification for the NASA International Space Station [13]. In a second project, engineers at Rockwell Aviation used the SCR tools to detect 24 errors, many of them serious, in the requirements specification of a flight guidance system [14]. A third of the detected errors were uncovered in constructing the specification, a third in running the consistency checker, and the remaining third in executing the the simulator. In a third project, NRL applied the SCR tools, to the specification of the Weapons Control Panel (WCP) of a military system. The tools uncovered a number of errors, including the violation of a critical safety property. Developing an SCR specification of WCP from the contractor specification, using the tools to detect specification errors, and building a working prototype of WCP from the specification required only one person-month, thus demonstrating the utility and cost-effectiveness of the SCR tools.

More than 200 organizations from academia, industry, and government have downloaded the SCR tools. The tools have been used in practice by companies such as Lockheed Martin for many years. Most recently, the tools were used to provide evidence to a certifying authority that a security-critical module of an embedded software device enforces data separation [15] and to specify the requirements of three safety-critical software modules of NASA systems [16].

Briefly described below are the ten tools that comprise the SCR Toolset. Four of the five tools shown in the box at the top of Fig. 2 have been distributed to external organizations. The fifth tool, the model checker Spin [17], can be obtained from Gerard Holzmann of NASA's Jet Propulsion Laboratory. For more details about the SCR Toolset, see [10].

Specification Editor

To create, modify, or display a requirements specification, the user invokes the SCR specification editor [10]. In the SCR method, each specification is organized into dictionaries and tables. The dictionaries define the static information in the specification, such as the names, values, and types of variables and constants; the user-defined types; etc. The tables define how the variables in the specification change in response to changes in monitored variables.

Consistency Checker

The SCR consistency checker [6], [10] checks for properties derived from the SCR requirements model. This tool detects syntax and type errors, incompleteness of variable definitions, missing cases, unwanted nondeterminism, and circular definitions. When an error is detected, the consistency checker provides detailed feedback to help the user correct the error. Consistency checking is a form of static analysis. Since it is accomplished without executing the specification or performing a reachability analysis, it is more efficient than model checking. When developing an SCR specification, the user normally invokes the consistency checker first and postpones more heavy-duty analysis such as model checking until later in development. By exploiting the special properties guaranteed by consistency checking, such as determinism, the later analyses can be more efficient [18].

Simulator

To validate a specification, the user can run the SCR simulator [10] and analyze the results to ensure that the specification captures the intended behavior. Additionally, the user can define invariant assertions believed to be true of the required system behavior and, using simulation, execute a series of scenarios to determine whether any violate the invariants. To provide input to the simulator, the user either enters a sequence of input events (changes in monitored variables) or loads a previously stored scenario.

The simulator supports alternative front-ends, tailored to particular application domains. For example, we have developed a customized front-end for pilots to use in evaluating an attack aircraft specification (see Fig. 3). Rather than clicking on monitored 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 sees results presented on a simulated cockpit display. This front-end allows the pilot to move out of the world of software requirements specification and into the world of attack aircraft,

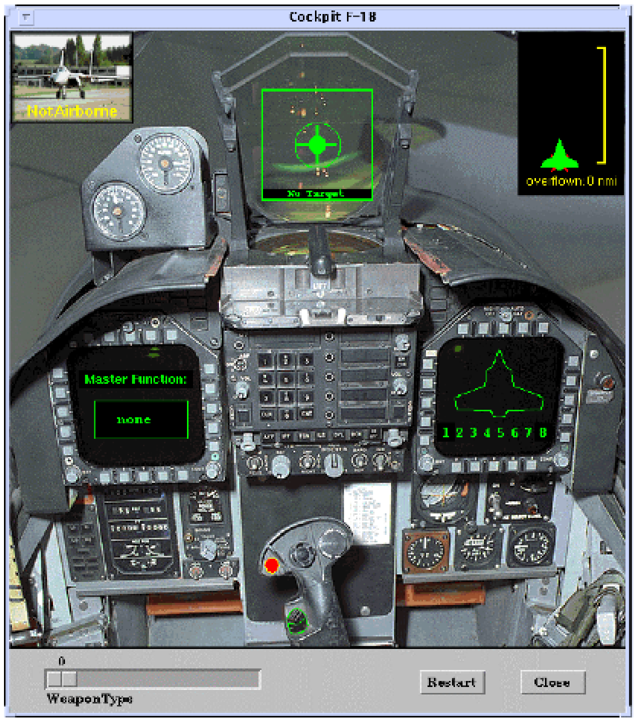

Fig. 3. Customized simulator front-end for an avionics system.

where he is the expert. Such an interface facilitates customer validation of the specification. A customized front-end, part of the working prototype mentioned above, has also been developed for the WCP.

Model Checker

The Spin model checker [17] has been integrated into the SCR toolset [18]. After using the tools described above to develop a formal requirements specification, a specifier can invoke Spin within the toolset to verify properties of the specification. Currently, we use Spin to analyze invariant properties. The SCR toolset automatically translates an SCR specification into *Promela*, the language of Spin. The user can demonstrate and validate a property violation detected by Spin with the SCR simulator.

The number of reachable states in a state machine model of a practical system is usually very large, sometimes infinite. To make model checking practical, we have developed sound methods for deriving abstractions from SCR specifications, based on the property to be analyzed [7], [18]. The methods are practical: none requires ingenuity on the user's part, and each derives a smaller, more abstract model. The SCR abstraction methods systematize techniques that users of model checkers routinely apply but often in ad hoc ways. These methods eliminate irrelevant variables as well as unneeded detail from the specification. For example, in analyzing a safety property for WCP, we used our abstraction methods to reduce the number of variables in the specification from 258 to 55, and to replace several real-valued variables with discrete variables, thus making model checking feasible (by making the state space finite).

Dependency Graph Browser

One attribute of SCR requirements specifications is that, while they give detailed information about specific aspects of the required system behavior, understanding the relationship between different parts of a specification can be difficult, especially for large specifications. To address this problem, a Dependency Graph Browser (DGB) has been developed, which represents the variable dependencies in a 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 invoke the DGB to display and extract subsets of the dependency graph, e.g., the subgraph containing all variables on which a selected controlled variable depends.

TAME Theorem Prover

TAME (Timed Automata Modeling Environment) [19], a specialized interface to PVS [2], offers templates for specifying automata models and customized strategies which implement high-level proof steps for proving automaton properties [20]. Initially developed for Timed Input/Output Automata, TAME has been adapted to SCR by an automatic SCR-to-TAME translator and by adding SCR-specific strategies that prove many properties automatically and exhibit 'problem transitions' for undischarged proof goals.

Invariant Generator

Algorithms for generating state invariants from SCR specifications are described in [21]. Such invariants are useful as auxiliary lemmas in proving properties of SCR specifications with TAME and Salsa. The SCR invariant generator generates invariants automatically. The user may choose which algorithms to apply and may also choose which tables (condition, event, or mode transition tables) to analyze.

Salsa Property Checker

The SCR property checker Salsa [22] may be used to check SCR specifications for Disjointness and Coverage and for satisfaction of state and transition invariants. Salsa can check the validity of formulas on Boolean, enumerated and integer variables restricted to Presburger arithmetic. It uses BDDs for analyzing formulas on Boolean and enumerated variables and an automata representation for analyzing Presburger arithmetic formulas.

Source Code Generator

While producing a high-quality requirements specification is crucial, ultimately software must be implemented to satisfy the requirements. A specification verified and validated using the SCR tools provides a solid basis for generating executable code. Although automatically generating code may be infeasible for some purposes (e.g. code that provides an interface to a physical device), such an approach is feasible for code that implements a program's control logic and simple data types. In such cases, the code can be automatically generated from an SCR requirements specification. Recently, we developed a grammar and a set of semantic rules for the SCR notation and used the APTS transformational system [23] to automatically generate C source code from an SCR requirements specification [24]. We have also developed a number of techniques for improving the efficiency of automatically generated code [25].

Test Case Generator

To convince customers that the implementation is acceptable and to detect errors, the software implementation must be tested. An enormous problem, however, is that software testing is extremely costly and time-consuming. Current estimates are that testing consumes between 40% and 70% of the software development effort [4]. We have developed a prototype Test Case Generator [26] which automatically constructs a suite of test cases from an SCR requirements specification. (A *test case* is a sequence of monitored variable changes, each coupled with a set of controlled variable changes.) To ensure that the test cases 'cover' all possible system behaviors, our technique organizes the set of possible system executions into equivalence classes and builds one or more test cases for each class. By reducing the human effort needed to build and run the test cases, this tool should reduce both the enormous cost and significant time and human effort characteristic of current software testing methods.

IV. Conclusions

Using a language such as SCR to specify software requirements has several advantages. First, due to its tabular notation, developers find an SCR requirements specification relatively easy to understand and the SCR notation relatively easy to apply in formulating requirements. Second, due to SCR's formal semantics, the specification of the required behavior is both unambiguous and precise. Finally, due to its formal state machine semantics, an SCR specification provides a sound basis for using formal techniques and tools to check the specifications for properties of interest. Like the SCR notation, the SCR tools are designed for software developers who lack advanced mathematical training and theorem proving skills. Hence, developers can use the tools to perform relatively complex formal analyses of requirements specifications. Given SCR's formal semantics coupled with its user-friendly design, the SCR language and tools provide a practical formal method for constructing a high quality requirement specification and for using that specification to automatically generate both source code and test cases.

References

[1] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: MIT Press, 1999.

[2] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, "PVS Prover Guide, Version 2.4," Computer Science Lab, SRI International, Menlo Park, CA, Tech. Rep., Nov. 2001.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *European Systems Conference*, 2006.

[4] B. Beckert, T. Hoare, R. Hahnle, D. R. Smith, C. C. Green, S. Ranise, C. Tinelli, T. Ball, and S. K. Rajamani, "Intelligent systems and formal methods in software engineering," *IEEE Intelligent Systems*, vol. 21, no. 6, pp. 73–85, 2006.

[5] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 1, pp. 2–13, Jan. 1980.

[6] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Transactions on Software Eng. and Methodology*, vol. 5, no. 3, pp. 231–261, 1996.

[7] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using abstraction and model checking to detect safety violations in requirements specifications," *IEEE Trans. on Softw. Eng.*, vol. 24, no. 11, Nov. 1998.

[8] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Trans. Softw. Eng.*, vol. 12, no. 2, pp. 251–257, 1986.

[9] C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj, "SCR*: A toolset for specifying and analyzing software requirements," in *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

[10] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords, "Tools for constructing requirements specifications: The SCR toolset at the age of ten," *Computer Systems Science and Engineering*, vol. 20, no. 1, pp. 19–35, Jan. 2005.

[11] D. L. Parnas and J. Madey, "Functional documentation for computer systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[12] P.-J. Courtois and D. L. Parnas, "Documentation for safety critical software," in *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, Baltimore, MD, 1993, pp. 315–323.

[13] S. M. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling." *IEEE Trans. Software Eng.*, vol. 24, no. 1, pp. 4–14, 1998.

[14] S. Miller, "Specifying the mode logic of a flight guidance system in CoRE and SCR," in *Proceedings of the 9th 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[15] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proc. 13th ACM Conf. on Comp. and Comm. Sec.*, 2006.

[16] C. Heitmeyer and R. Jeffords, "Applying a formal requirements method to three NASA systems: Lessons learned," in *Proc. 2007 IEEE Aerospace Conf.*, 2007.

[17] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[18] R. Bharadwaj and C. Heitmeyer, "Model checking complete requirements specifications using abstraction," *Automated Software Engineering*, vol. 6, no. 1, Jan. 1999.

[19] M. Archer, "TAME: Using PVS strategies for special-purpose theorem proving," *Annals of Mathematics and Artificial Intelligence*, vol. 29, no. 1-4, pp. 131–189, Feb. 2001.

[20] M. Archer, C. Heitmeyer, and E. Riccobene, "Proving invariants of I/O automata with TAME," *Automated Software Engineering*, vol. 9, pp. 201–232, 2002.

[21] R. Jeffords and C. Heitmeyer, "Automatic generation of state invariants from requirements specifications," in *Proc., 6th ACM SIGSOFT Internat. Symp. on Foundations of Software Engineering (FSE '98)*, November 1998.

[22] R. Bharadwaj and S. Sims, "Salsa: Combining constraint solvers with BDDs for automatic invariant checking," in *Proc, 6th Internat. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, S. Graf and M. Schwartzbach, Eds., vol. 1785. Springer, April 2000, pp. 378–394.

[23] R. Paige, "APTS external specification manual (rough draft)," New York University, Available at http://www.cs.nyu.edu/˜jessie/, Tech. Rep., 1993, unpublished manuscript.

[24] E. I. Leonard and C. L. Heitmeyer, "Program synthesis from formal requirements specifications using APTS," *Higher-Order and Symbolic Computation*, vol. 16, no. 1-2, pp. 63–92, 2003.

[25] T. Rothamel, C. L. Heitmeyer, E. I. Leonard, and Y. A. Liu, "Generating optimized code from scr specifications." in *LCTES*, 2006, pp. 135–144.

[26] A. Gargantini and C. Heitmeyer, "Automatic generation of tests from requirements specifications," in *Proc. ACM 7th Eur. Software Eng. Conf. and 7th ACM SIGSOFT Symp. on the Foundations of Software Eng. (ESEC/FSE99)*, Toulouse, FR, Sep. 1999.