Software Cost Reduction

Constance L. Heitmeyer[*]

Introduction

Software Cost Reduction (SCR) is a set of techniques for designing software systems developed by David Parnas and researchers from the U.S. Naval Research Laboratory (NRL) beginning in the late 1970s. A major goal of the original SCR research team was to evaluate the utility and scalability of software engineering principles by applying the principles to the reconstruction of software for a practical system, the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft. The process of applying the principles produced a number of new techniques for software design, which were demonstrated in a requirements document [18] and several software design documents (e.g., a module guide [6]) for the A-7. Further research during the 1990s produced two formal models, the Four Variable Model [37] and the SCR requirements model [15], and a set of software tools for analyzing SCR-style requirements documents [16].

A central notion of SCR is that software should be designed using an idealized process called the "Rational Design Process" [36]. Although designing software using a perfectly rational process is impossible, software developers are more likely to produce a rational design if they follow a rational process rather than if they proceed on an ad hoc basis. In the Rational Design Process, software is designed and implemented in stages. At each stage, a work product, such as a requirements document or a design document, is produced. Each work product is associated with criteria that the work product must satisfy and a description of the information that the work product contains. This article focuses on the SCR techniques for constructing and evaluating the requirements document, the work product built during the requirements stage of software development, and the aspect of SCR that has received significant attention during both the early and the more recent research. It also briefly describes, and gives pointers to, the SCR approach to software design, focusing on the design and documentation of the module structure, the module interfaces, and the uses hierarchy.

[*]Code 5546, Naval Research Laboratory, Washington, DC 20375. This work was supported by the Office of Naval Research.

SCR Approach to Requirements

The A-7 Requirements Document

The A-7 requirements document, a fully worked-out specification of the required behavior of the A-7 OFP, was published in 1978 to demonstrate the SCR techniques for specifying software requirements [18, 19, 1]. This document introduced three major aspects of the SCR approach to requirements: the focus on outputs, a special tabular notion for specifying each output, and a set of criteria for evaluating a requirements document. A critical step in constructing an SCR software requirements document is to identify all outputs that the software must produce and to express the value of each output as a mathematical function of the state and history of the environment. To represent these functions precisely and compactly, the A-7 document introduced a special tabular notation. This notation facilitates writing and understanding the functions and also aids in detecting specification errors, such as missing cases and ambiguity. In addition to specifying the outputs, the A-7 document contained a specification of the input and output interfaces that the software would use to communicate with its environment, a specification of the computers on which the software was expected to run, timing and accuracy constraints on each output, a description of ways in which the software was likely to change, and a discussion of software responses to undesired events (e.g., the failure of an input or output device). To be acceptable, a requirements document must satisfy selected criteria, including completeness (i.e., any implementation satisfying every statement in the requirements document should be acceptable), freedom from implementation bias, and organization as a reference document (information in the document should be easy to find).

To specify the required behavior precisely and concisely, the A-7 requirements document introduced conditions, events, modes, and terms [19]. A *condition* was defined as a predicate that characterizes "some aspect of the system for a measurable period of time." An *event* occurs when the value of a condition changes from true to false or vice versa. The notation "@T(c)" was introduced to denote that condition c becomes true and "@F(c)" to denote that c becomes false. A *mode* was defined as a class of system states and a *term* as a "text macro" that reduces redundancy.

During the 1980s and 1990s, a number of organizations in both industry and government (e.g., Grumman, Bell Laboratories, NRL, and Ontario Hydro), used the SCR techniques to document the requirements of a wide range of practical systems, including the OFP for the A-6 aircraft [27], the Bell telephone network [20], a submarine communications system [17], and safety-critical components of the Darlington nuclear power plant [32, 26]. Moreover, in 1994, a version of SCR called CoRE [11] was used to document the requirements of the OFP of Lockheed's C-130J aircraft [12]. The Lockheed requirements document contains over 1000 tables and the OFP over 250K lines of Ada source code, thus demonstrating that the SCR techniques scale.

The Four Variable Model

To generalize the SCR techniques for writing requirements and to establish a formal foundation for the notions introduced in the A-7 requirements document (e.g., the inputs and outputs, the accuracy and timing requirements, and the required responses to undesired events), Parnas and Madey in 1995 published the Four Variable Model [37]. This model represents the required behavior of a software system in terms of four sets of variables—monitored, controlled, input, and output variables—and four relations—NAT, REQ, IN, and OUT. Whereas the A-7 requirements document specifies the required behavior of *software* by describing outputs as functions of the state and history of the environment, the Four Variable Model describes the required behavior of a *software system* by describing the required relation between two sets of environmental quantities, quantities that the system monitors and those that it controls. NAT and REQ are relations on the *monitored* and *controlled variables*, variables that represent the time-varying discrete and continuous environmental quantities that the system monitors and controls. NAT describes *assumptions* about system behavior, i.e., constraints imposed on the monitored and controlled quantities by physical laws and the system environment. REQ describes those aspects of the environment that the system is expected to control, i.e., how the system is required to change the controlled quantities in response to changes in the monitored quantities.

In the Four Variable Model, the system requirements are specified in two steps. First, the "ideal" system behavior is specified: i.e., NAT and REQ are defined as if the system could obtain perfect values of the monitored variables and compute perfect values of the controlled variables. Next, the relations IN and OUT are used to specify the tolerances, i.e., the accuracy required in measuring values of the monitored quantities and in computing values of the controlled quantities. In the model, input devices (e.g., sensors) measure values of the monitored quantities and output devices (e.g., actuators) assign values to the controlled quantities. The variables that the input devices read, called *input variables*, and those that the output devices write, called *output variables*, are directly available to the software.[1] IN defines the tolerances on the monitored quantities as a mapping from the monitored quantities to the input variables. Similarly, OUT defines the the tolerances on the controlled quantities as a mapping from the output variables to the controlled quantities.

Example

To illustrate the SCR approach to requirements, this section introduces a simple control system which turns safety injection on and off in a nuclear power plant. This Safety Injection System (SIS), a simplified version of the system described in [9], monitors water pressure and adds coolant to the reactor core when the pressure falls below some threshold. A drop in water pressure below the constant

[1]These variables correspond to the inputs and outputs in the A-7 requirements document.

Low causes the SIS to enter mode TooLow; an increase in water pressure above a larger constant Permit causes the system to enter mode High. A system operator blocks safety injection by turning a “Block” switch to On and resets the SIS after blockage by turning a “Reset” switch to On. An assumption in the SIS is that water pressure ranges between 0.0 and 2000.0 psi (pounds per square inch).

The SIS requirements may be represented in SCR using three monitored variables, a controlled variable, a mode class (a set of modes), and a term. The monitored variables WaterPres, Block, and Reset and the controlled variable SafetyInjection represent the three monitored quantities and the single controlled quantity. The mode class Pressure contains three modes, TooLow, Permitted, and High, each representing a range of values of the monitored variable WaterPres. The term Overridden describes when safety injection is blocked. Each of these state variables (a *state variable* is a monitored or controlled variable, a mode class, or a term) may be used to describe conditions and events. An example of a condition in the SIS specification is “WaterPres < Low”. Two types of events are *monitored events*, events that occur when a monitored variable changes value, and *conditioned events*, events that occur when a specified condition is true. In the SIS specification, an example of a monitored event is “@T(Block=On)” (the operator turns Block from Off to On); an example of a conditioned event is “@T(Block=On) WHEN WaterPres < Low” (the operator turns Block to On when water pressure is below Low).

Different parts of the above description may be associated with the REQ and NAT relations. To define REQ, the values of the three dependent variables, Pressure, Overridden, and SafetyInjection, are expressed as mathematical functions. Composing these functions defines REQ, the required relation (in this example, a function) between the monitored and controlled variables. The assumptions, WaterPres is a non-negative real-valued variable no greater than 2000.0 and Low < Permit, are considered part of NAT. The SIS example may also be used to illustrate the IN relation. Suppose three sensors measure water pressure, and let the input variable w_i, $1 \leq i \leq 3$, represent the value read by sensor i. If each sensor is required to measure WaterPres within one psi, then the predicate $|\texttt{WaterPres} - w_i| \leq 1$ is part of IN. Alternately, if the average of the values read by the three sensors must be within one psi of the actual value of WaterPres, then $|\texttt{WaterPres} - (w_1 + w_2 + w_3)/3| \leq 1$ is part of IN.

SCR Tables

Among the tables in SCR specifications are condition tables, event tables, and mode transition tables. Each defines a dependent variable (a controlled variable, mode class, or term) as a mathematical function. Typically, a condition table defines a variable as a function of a mode and a *condition*, and an event table defines a variable as a function of a mode and an *event*. A mode transition table, a special case of an event table, defines a mode as a function of a mode and an event. In some cases, a condition or event table is modeless, i.e., defines a variable without referring to modes.

Tables 1–3 define REQ, the required relation between the monitored and controlled variables in the SIS. Table 1 is a mode transition table describing the mode class `Pressure` as a function of the current mode and events defined on the monitored variable `WaterPres`. The table makes explicit all events that change the value of `Pressure`. For example, the first row states, "If `Pressure` is `TooLow` and `WaterPres` rises to or above `Low`, then `Pressure` changes to `Permitted`." Events which do not change the mode are omitted from the table. For example, if `Pressure` is `TooLow` and `WaterPres` changes but remains less than `Low`, then `Pressure` remains `TooLow` after the event.

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

Table 1: Mode Transition Table for `Pressure`.

Table 2 is an event table describing the term `Overridden` as a function of `Pressure` and the monitored variables `Block` and `Reset`. Like mode transition tables, event tables make explicit only those events that cause the variable defined by the table to change. For example, the middle entry in the second row states, "If `Pressure` is `TooLow` or `Permitted` and `Block` becomes `On` when `Reset` is `Off`, then `Overridden` becomes *true*." In contrast, if the mode is `High` and either `Block` or `Reset` changes, then the value of `Overridden` remains the same because no events in the first row involve either `Block` or `Reset`. The entry "False" in row 1 means that when the mode is `High`, no event can cause `Overridden` to become *true*.

Mode Class Pressure	**Events**	
High	False	@F(Pressure=High)
TooLow, Permitted	@T(Block=On) WHEN Reset=Off	@T(Pressure=High) OR @T(Reset=On)
Overridden	True	False

Table 2: Event Table for `Overridden`.

Table 3 is a condition table describing the controlled variable `SafetyInjection` as a function of `Pressure` and the term `Overridden`. Table 3 states, "If `Pressure` is `High` or `Permitted`, or if `Pressure` is `TooLow` and `Overridden` is *true*, then `Safety Injection` is `Off`; if `Pressure` is `TooLow` and `Overridden` is *false*, then `Safety Injection` is `On`." The entry "False" in the first row means that `Safety Injection` is never `On` when `Pressure` is `High` or `Permitted`.

Mode Class `Pressure`	**Conditions**	
`High`, `Permitted`	True	False
`TooLow`	`Overridden`	NOT `Overridden`
`Safety Injection`	`Off`	`On`

Table 3: Condition Table for `Safety Injection`.

SCR Requirements Model

The purpose of the SCR requirements model [15] is two-fold: to assign a precise semantics to the constructs and notation in SCR requirements specifications and to provide a formal foundation for mechanized analysis of the specifications. A special case of the Four Variable Model, the SCR model represents a system as a state machine and focuses on the REQ and NAT relations. Representing a system as a state machine means that SCR requirements specifications based on this model usually assign monitored and controlled variables discrete values. For example, using this model, the variable `WaterPres` in the SIS could be represented as a non-negative integer that does not exceed 2000.

In the model, a *system state* is defined as a function mapping each state variable to a type-correct value and TY as a function that maps each state variable to its type, i. e., set of legal values. In the SIS, the type definitions include

$$\begin{aligned}
\mathrm{TY}(\texttt{Pressure}) &= \{\texttt{TooLow, Permitted, High}\} \\
\mathrm{TY}(\texttt{WaterPres}) &= \{0, 1, 2, \cdots, 2000\} \\
\mathrm{TY}(\texttt{Overridden}) &= \{\textit{true, false}\} \\
\mathrm{TY}(\texttt{Block}) &= \{\texttt{On, Off}\}.
\end{aligned}$$

In the model, a *condition* is a predicate on a single system state and an *event* a predicate on two system states which is true if the states differ in the value of at least one state variable. The model defines a conditioned event "@T(c) WHEN d" as

$$@\mathrm{T}(c)\ \mathrm{WHEN}\ d \overset{\mathrm{def}}{=} \neg c \ \wedge\ c' \ \wedge\ d, \tag{1}$$

where c and d are conditions, and the unprimed c denotes c in the old state and the primed c denotes c in the new state. Applying the definition in (1), the conditioned event @T(`Block`=`On`) WHEN `Reset`=`Off` can be rewritten as $\texttt{Block} \neq \texttt{On} \ \wedge\ \texttt{Block}' = \texttt{On} \ \wedge\ \texttt{Reset} = \texttt{Off}$. This event occurs if both `Block` and `Reset` are `Off` in the old state and `Block` is switched `On` in the new state.

In the SCR model, a software system Σ is represented as a state machine $\Sigma = (S, S_0, E^m, T)$, where S is a set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of monitored events, and T is the transform describing the allowed state transitions. The transform T (which corresponds to REQ in the Four Variable Model) maps a monitored event e in E^m and a state s in S to a new state s'. A basic assumption, called the *One Input Assumption*, is that exactly one monitored event occurs at each state transition. A second assumption, called the *Synchrony Assumption*, requires a system Σ to completely process each monitored event before it processes the next monitored event. To compute the next state, the transform T composes the functions derived from the condition, event, and mode transition tables. For T to be well-defined, no circular dependencies are allowed in the definitions of the state variables. To achieve this, the model requires a partial order on the values of state variables in the new state.

The model contains definitions of the functions that can be derived from the SCR tables.[2] Applying the definition in the model to the condition table in Table 3 produces the following definition of the controlled variable `SafetyInjection`:

$$\mathtt{SafetyInjection} = \begin{cases} \mathtt{Off} & \text{if} \quad \mathtt{Pressure} = \mathtt{High} \vee \mathtt{Pressure} = \mathtt{Permitted} \vee \\ & \qquad (\mathtt{Pressure} = \mathtt{TooLow} \wedge \mathtt{Overridden} = \mathit{true}) \\ \mathtt{On} & \text{if} \quad \mathtt{Pressure} = \mathtt{TooLow} \wedge \mathtt{Overridden} = \mathit{false}. \end{cases}$$

Similarly, applying the definition in the model to the event table in Table 2 produces the following definition of the term `Overridden`:

$$\mathtt{Overridden}' = \begin{cases} \mathit{true} & \text{if} & (\mathtt{Pressure} = \mathtt{TooLow} \wedge \mathtt{Block}' = \mathtt{On} \wedge \\ & & \mathtt{Block} = \mathtt{Off} \wedge \mathtt{Reset} = \mathtt{Off}) \vee \\ & & (\mathtt{Pressure} = \mathtt{Permitted} \wedge \mathtt{Block}' = \mathtt{On} \wedge \\ & & \mathtt{Block} = \mathtt{Off} \wedge \mathtt{Reset} = \mathtt{Off}) \\ \mathit{false} & \text{if} & (\mathtt{Pressure} = \mathtt{TooLow} \wedge \mathtt{Reset}' = \mathtt{On} \wedge \\ & & \mathtt{Reset} = \mathtt{Off}) \vee \\ & & (\mathtt{Pressure} = \mathtt{Permitted} \wedge \mathtt{Reset}' = \mathtt{On} \wedge \\ & & \mathtt{Reset} = \mathtt{Off}) \vee \\ & & (\mathtt{Pressure}' = \mathtt{High} \wedge \mathtt{Pressure} \neq \mathtt{High}) \\ \mathtt{Overridden} & \text{otherwise} & \end{cases}$$

To define the required behavior completely and unambiguously, each SCR table must define a total function. To achieve this, the model requires the information in each table to satisfy certain properties. To define the required behavior unambiguously, each condition and event table must satisfy the Disjointness Property: the pairwise conjunction of conditions (events) in each row of a condition (an event) table must always be false. Inspection of Tables 2 and 3 shows that both tables satisfy the Disjointness Property. For example, in Table 3, $\mathit{true} \wedge \mathit{false} = \mathit{false}$ and $\mathtt{Overridden} \wedge \neg\mathtt{Overridden} = \mathit{false}$. To define the required behavior completely, a condition table must satisfy the Coverage Property: the disjunction of the conditions in each row of the table must be

[2]For a more general model of tabular expressions, see [22].

true. Inspection shows that the condition table in Table 3 satisfies the Coverage Property (since $\mathit{true} \lor \mathit{false} = \mathit{true}$ and $\texttt{Overridden} \lor \neg\texttt{Overridden} = \mathit{true}$). By requiring the value of the variable defined by an event table to remain the same if an event occurs which does not appear explicitly in the table, the model ensures that the table defines a total function.

SCR Tools

Until the mid-1990s, SCR requirements specifications were analyzed for defects using mostly manual inspection. Although inspection can expose many defects, it has two serious shortcomings. First, inspection can be very expensive. In the certification of the Darlington system, for example, the inspection of SCR tables cost millions of dollars. Second, human inspections often overlook many errors. In a 1996 study by NRL, for example, a mechanized analysis of the condition tables and mode transition tables in the A-7 requirements document exposed 17 missing cases (violations of Coverage) and 57 instances of ambiguity (violations of Disjointness) [15]. These flaws were detected even though the document had previously been inspected by two independent review teams. In a 1998 study by Rockwell Aviation, which produced similar results, software tools exposed 28 errors, many of them serious, in a requirements specification of a flight guidance system [29]. The discovery of so many errors was somewhat surprising given that the specification, according to the project leader [28], "represented our best effort at producing a correct specification manually."

While human effort is critical to creating specifications and manual inspections can detect many specification errors, effective use of SCR requirements specifications in industrial settings requires automated tool support. Not only can tools find specification errors that inspections miss, they can do so more cheaply. To explore what form such tools should take, NRL has designed a suite of software tools for constructing and analyzing requirements specifications in the SCR tabular notation.

To develop an SCR requirements specification, a four-step process may be followed. Like the Rational Design Process, this is an idealization of a real-world process. First, a requirements specification is constructed using the SCR tabular notation. Second, the specification is analyzed for violations of application-independent properties, such as missing cases and unwanted ambiguity. Third, the specification is validated by application experts to ensure that it captures the intended behavior. Finally, the specification is analyzed for critical application properties, such as security and safety properties.

The SCR tools may be used to support this process. To begin, the user invokes a tool called the specification editor to construct the SCR requirements specification [13]. Next, the user invokes the consistency checker [15] to analyze the specification for properties derived from the SCR requirements model. Designed to detect errors automatically, the consistency checker exposes syntax and type errors, variable name discrepancies, missing cases, ambiguity, and circular definitions. When an error is detected, the consistency checker provides detailed feedback to aid in error correction. To perform the most computationally complex checks, checks for Disjointness (to detect nondeterminism) and for Coverage (to detect missing cases), the consistency checker uses an extension of the semantic tableaux algorithm [38].

To validate the specification, the user may run *scenarios*, sequences of monitored events, through the SCR simulator [13, 16] and analyze the results to ensure that the specification captures the intended behavior. In addition, the user can define application properties believed to be true of the required behavior and, using simulation, execute a series of scenarios to determine if any violate the properties. To facilitate validation of the specification by application experts, the simulator supports the rapid construction of graphical user interfaces, customized for particular application domains [16].

To analyze an SCR requirements specification for application properties, the user may first run a model checker, such as SPIN [21], to analyze a finite state model of the specification. Prior to model checking, the SCR tools automatically translate the specification into Promela, the language of SPIN. Often, model checking exposes property violations. Due to the state explosion problem, model checking may not be effective for verifying application properties. To verify properties of an SCR specification, the user may apply a theorem prover, such as TAME [2, 3], a specialized interface to the general-purpose theorem prover PVS [30], or Salsa, an automatic theorem prover based on decision procedures [4], to prove properties automatically. In using either TAME or Salsa to verify application properties, completing a proof may require auxiliary lemmas. To construct candidate lemmas, the user may invoke the SCR invariant generator [23, 24], which automatically generates *state invariants*, properties true of every reachable state, from an SCR requirements specification.

Applying the SCR Tools to Practical Systems

The SCR tools have been applied in four projects involving practical systems. In one project, NASA used the consistency checker to detect missing assumptions and unwanted nondeterminism in the requirements specification of the International Space Station [10]. In a second project, Rockwell Aviation used the SCR tools to detect 28 errors in the requirements specification of a flight guidance system (FGS) [29]. One-third of the errors were found when the FGS requirements specification was entered into the SCR toolset, another third when the consistency checker was applied, and the remaining third when the simulator was executed. These results suggest that different tools find different classes of errors. In a third project, NRL used the SCR tools to expose a serious safety violation in a moderately large contractor specification of a U.S. weapons system [14]. This specification, which contains over 250 variables and six safety properties, was translated semi-automatically into the SCR tabular notation. Applying abstraction to reduce the size of the SCR specification and then invoking SPIN on the abstract model exposed the safety violation. In a fourth project, NRL used the SCR tools to analyze the requirements specification of a cryptographic device for eight security properties [25]. Individually, both TAME and Salsa automatically verified seven of the properties. In proving three of the properties, state invariants constructed by the invariant generator were required. TAME and Salsa also detected a possible violation of the eighth property. Experimentation with the simulator validated that the detected problem was an actual violation.

SCR Approach to Software Design

Introduced briefly below are three major activities in the SCR approach to software design and the documentation associated with each. For more information about the overall SCR approach to software design, see [36].

Designing the Module Structure

A critical activity in the design of software is the decomposition of the software into modules. In the SCR approach, each module is either a collection of submodules or a single work assignment, that is, a programming task that can be completed by a single programmer. The overall goal of module decomposition is to reduce the cost of software development and maintenance by allowing modules to be designed, implemented, and modified independently. In SCR, this is achieved by applying the information hiding principle [31] to module decomposition. According to this principle, the system details that are likely to change independently are assigned to separate modules. To begin, the information hiding principle is used to divide the software into a small number of modules. Then, information hiding is used to decompose each of these modules into submodules. This process continues until each module is small enough to describe a single work assignment.

The document which describes the module structure is called the module guide. This guide, which has a tree structure, describes the responsibility of each module by stating the design decisions that will be encapsulated by the module. For a complete example of a module guide, see [6]. For more information about the SCR approach to designing the module structure, see [7, 33].

Designing the Module Interfaces

To construct software efficiently, programmers must be able to work independently. Although the module guide describes the responsibility of each module, it does not provide sufficient information for the programmer who will implement the module nor for other programmers whose modules will use the module. The purpose of the module interface is to describe 1) the set of *assumptions* that the programmers responsible for other modules may make about the module and

2) the set of *access programs* that programs in other modules use to access the data or services provided by the module. Two major classes of access programs exist: those that return information to the calling program and those that change the state of the module to which the program belongs.

For each module, a module interface specification must be written. This specification must be formal and provide a blackbox view of the module. The module interface specification should be reviewed by programmers who will use the module as well as others interested in the design, e.g., reviewers. For more information about the design of module interfaces, see [5, 8].

Designing the Uses Hierarchy

Once the modules and their interfaces are known, the uses hierarchy [35] can be defined. This hierarchy is a relation defined on the access programs in the module interface specifications. Suppose A and B are access programs. Then, we say that A *uses* B if and only if the correctness of program A depends on the presence of a correct program B. Requiring the access programs to be organized by a uses hierarchy, i.e., a loop-free graph, eliminates interdependencies among the programs.

Restricting the uses relation to a loop-free graph has the following advantage: it defines a number of usable subsets of the complete system. Suppose level 0 of the uses hierarchy is associated with the set of programs that use no other programs and that level i, $i \geq 1$, is associated with the set of programs that use at least one program at level $i - 1$ and no program at a level higher than $i - 1$. Then, each level of the hierarchy is associated with a usable subset of the system. This avoids the problem of many systems in which nothing works unless everything works. Usable subsets are not only important for staged deliveries, they are also important in the development of program families [34]. A convenient way to document the uses hierarchy is to use a binary matrix, where the entry in position (A, B) is *true* if and only if program A uses program B. For more information about the uses hierarchy and an example, see [35].

References

[1] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Lab., Wash., DC, 1992.

[2] M. Archer, C. Heitmeyer, and E. Riccobene. Using TAME to prove invariants of automata models: Case studies. In *Proc. 2000 ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP'00)*, August 2000.

[3] Myla Archer. Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000.

[4] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '2000)*, Berlin, March 2000.

[5] Kathryn Britton, R. A. Parker, and David Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proc. 5th Int'l Conf. on Softw. Eng. (ICSE '81)*, 1981.

[6] Kathryn Britton and David L. Parnas. A-7E software module guide. Technical Report 4702, Naval Research Lab., Wash., DC, 1981.

[7] Paul Clements, David Parnas, and David Weiss. Enhancing reusability with information hiding. In *Proc. Workshop on Reusability in Programming*, pages 240–247, September 1983. Available at http://chacs.nrl.navy.mil/publications/a7/.

[8] Paul C. Clements, R. A. Parker, David L. Parnas, John Shore, and Kathryn Britton. A standard organization for specifying abstract interfaces. Technical Report 8815, Naval Research Lab., Wash., DC, 1984. Available at http://chacs.nrl.navy.mil/publications/a7/.

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

[10] Steve Easterbrook, Robyn Lutz, Richard Covington, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1), January 1998.

[11] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[12] S. R. Faulk, L. Finneran, J. Kirby, Jr., S. Shah, and J. Sutton. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94)*, Gaithersburg, MD, June 1994.

[13] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 109–122, Gaithersburg, MD, June 1995.

[14] 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.*, 24(11), November 1998.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[16] Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

[17] Constance L. Heitmeyer and John McLean. Abstract requirements specifications: A new approach and its application. *IEEE Trans. Softw. Eng.*, SE-9(5):580–589, September 1983.

[18] Kathryn Heninger, David L. Parnas, John E. Shore, and John W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[19] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, January 1980.

[20] S. D. Hester, D. L. Parnas, and D. F. Utter. Using documentation as a software design medium. *Bell System Tech. J.*, 60(8):1941–1977, October 1981.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[22] Ryszard Janicki and Ridha Khedri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 39(2-3):189–213, March 2001.

[23] Ralph Jeffords and Constance Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, November 1998.

[24] Ralph D. Jeffords and Constance L. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proc. Fifth IEEE Int'l Symp. on Requirements Engineering (RE'01)*, Toronto, Canada, August 2001.

[25] J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*. IEEE Computer Society Press, December 1999.

[26] M. Lawford, J. McDougall, P. Froebel, and G. Moum. Practical application of functional and relational methods for the specification and verification of safety critical software. In *Proc. Algebraic Methodology and Software Technology, 8th Intern. Conf. (AMAST 2000), LNCS 1816*, Iowa City, Iowa, May 2000.

[27] S. Meyers and S. White. Software requirements methodology and tool study for A6-E technology transfer. Technical report, Grumman Aerospace Corp., Bethpage, NY, July 1983.

[28] S. P. Miller, March 1997. Personal communication.

[29] Steve Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd ACM Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[30] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[31] D. L. Parnas. Technique for software module specification with examples. *Communications of the ACM*, 15(5), 1972.

[32] D. L. Parnas, G.J.K. Asmis, and Jan Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2), April–June 1991.

[33] David Parnas, Paul Clements, and David Weiss. The modular structure of complex systems. In *Proc. 7th Int'l Conf. on Softw. Eng. (ICSE '84)*, pages 408–417, March 1984.

[34] David L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2, 1976.

[35] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5(2):128–138, February 1979.

[36] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng.*, SE-12(2):251–257, February 1986.

[37] David L. Parnas and Jan Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[38] R. M. Smullyan. *First-Order Logic.* Springer-Verlag, 1968. Republished by Dover Publications Inc., 1993.