Automated Consistency Checking of Requirements Specifications

Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw*

To appear in *ACM Trans. on Software Eng. and Methodology 5,* 3, July 1996, 231-261.

Abstract

This paper describes a formal analysis technique, called *consistency checking*, for automatic detection of errors, such as type errors, nondeterminism, missing cases, and circular definitions, in requirements specifications. The technique is designed to analyze requirements specifications expressed in the SCR (Software Cost Reduction) tabular notation. As background, the SCR approach to specifying requirements is reviewed. To provide a formal semantics for the SCR notation and a foundation for consistency checking, a formal requirements model is introduced; the model represents a software system as a finite state automaton, which produces externally visible outputs in response to changes in monitored environmental quantities. Results are presented of two experiments which evaluated the utility and scalability of our technique for consistency checking in a real-world avionics application. The role of consistency checking during the requirements phase of software development is discussed.

1 Introduction

Errors in requirements are pervasive, dangerous, and costly [13]. It is well known that the majority of software errors are introduced during the requirements phase [49]. There is also growing evidence that requirements errors can lead to serious accidents. For example, a 1992 study found that the major source of safety-related software errors in NASA's Voyager and Galileo spacecraft were errors in functional and interface requirements [37]. Unfortunately, fixing requirements errors can be extremely costly, especially if the errors are detected late in the software life-cycle. It is estimated that correcting requirements errors late (e.g., during maintenance) can cost up to 200 times as much as correcting

*Code 5546, Naval Research Laboratory, Washington, DC 20375. This paper is an extended version of a paper, "Consistency Checking of SCR-Style Requirement Specifications," published in the *Proceedings of the Second IEEE International Symposium on Requirements Engineering,* York, England, March 27-29, 1995. This work was supported by ONR and SPAWAR.

the errors during the requirements phase [5, 11]. Given the high frequency of requirements errors, the serious accidents they may cause, and the high cost of correcting them late, techniques for improving the quality of requirements documents and for early detection of requirements errors are crucial.

One promising approach to reducing requirements errors is to apply formal methods during the requirements phase of software development. By a *formal method*, we mean a development method based on some formalism, such as a formal specification notation or a formal analysis technique. A formal requirements specification can reduce errors by reducing ambiguity and imprecision and by making some instances of inconsistency and incompleteness obvious. Formal analysis can detect many classes of errors in requirements specifications, some of them automatically.

The SCR (Software Cost Reduction) requirements method was introduced more than a decade ago to specify the software requirements of real-time embedded systems unambiguously and concisely [26, 27]. Recently, the method has been extended to describe *system*, rather than simply *software*, requirements and to incorporate both functional requirements (the values the system assigns to outputs) *and* nonfunctional (e.g., timing and accuracy) requirements [45, 50, 51]. Recent work has also strengthened the method's formal underpinnings [12, 50, 45].

Designed for use by engineers, the SCR method has been successfully applied to a variety of practical systems, including avionics systems, such as the A-7 Operational Flight Program [26, 1]; a submarine communications system [25]; and safety-critical components of the Darlington nuclear power plant in Canada [51]. More recently, a version of the SCR method called CoRE [14] was used to document requirements of Lockheed's C-130J Operational Flight Program (OFP) [15]. The OFP consists of more than 100K lines of Ada code, thus demonstrating the scalability of the SCR method.

While the above applications of SCR rely on manual techniques, effective use of the method in industrial settings will require powerful and robust tool support. A significant barrier to industrial use of formal methods to date has been the weakness of the methods associated with given formalisms. Although much attention has been focused on the *formal* aspects of formal methods, too little effort has been devoted to the supporting methods. To be useful in developing practical systems, not only must formal methods provide rigor, in addition they must be supported by robust, well-engineered tools. In many practical cases, a large amount of detail is required to apply a formal method. This detail is unmanageable without some automation.

To provide automated support for the SCR requirements method, we are developing a suite of prototype tools for constructing and analyzing formal requirements specifications [20]. The tools include a *specification editor* for creating and modifying the specifications, a *simulator* for symbolically executing the system based on the specifications, and *formal analysis* tools for checking the specifications for selected properties.

One analysis tool, called a *consistency checker*, checks a requirements specification for properties defined by our formal requirements model [24]. Because the requirements model describes properties that all SCR requirements specifications must satisfy, the properties checked by the consistency checker are independent of a particular application. A second analysis tool, called a *verifier*, checks the specification for critical application properties, such as timing properties [22] and security properties [34]. Because verification of application properties depends on a consistent requirements specification, analysis using a verifier logically follows analysis with a consistency checker.

Checking the consistency of an SCR requirements specification is usually quite simple. For example, given a specification that includes a total function F, the consistency checker analyzes F to make sure it is total (i.e., defined everywhere in F's domain). Although checking such properties is usually straightforward, the number of times the properties need to be checked in practical requirements specifications can become very large, and thus reviewers must spend considerable time and effort verifying that the specifications have the properties. In fact, in the certification of the Darlington plant, Parnas has observed that the "reviewers spent too much of their time and energy checking for simple, application-independent properties" (such as the ones we describe in this paper) which distracted them from the "more difficult, safety-relevant issues" [41]. Tools that automatically perform such checks can save reviewers considerable time and effort, liberating them to do more creative work.

An industrial-strength formal method should be usable by engineers, scalable, and cost-effective. Automated consistency checking as described in this paper is an important step in developing such a method for requirements specification. It is easy to use: after developing a requirements specification in the SCR notation, a developer invokes the consistency checker to find inconsistencies automatically. It scales up to handle practical applications: in two experiments, our automated consistency checker found significant errors (that is, missing cases and nondeterminism) in the requirements specification of a medium-size Navy application. These errors were detected even though the specification had previously undergone systematic, comprehensive checks by two independent review teams. These results and the high cost (several million dollars) of the Darlington certification effort, where such checks were done by hand, suggest that automated consistency checking is more cost-effective than manual techniques.

After reviewing the SCR method for specifying requirements, this paper introduces our formal requirements model, describes consistency checks based on the model, presents the results of experiments we conducted to determine the utility of automated consistency checking, and discusses the role of consistency checking in the software development process. The contributions of this paper include 1) formal definition and application of a class of analysis, which we call consistency checking, for detecting domain-independent errors in requirements specifications and 2) evidence that software tools for automated consistency checking are usable, scalable, and cost-effective.

2 Review of the SCR Method

Background. The purpose of a requirements document is to describe all acceptable system implementations [25]. The software requirements document for the A-7 aircraft's Operational Flight Program was published in 1979 to demonstrate a systematic approach to producing such a document. The A-7 document introduces many features associated with the SCR requirements method—the tabular notation, the underlying finite state machine model, and special constructs for specifying requirements, such as conditions and events, input and output data items, mode classes, and terms. Recently, a number of researchers, including Faulk [12, 14, 15], van Schouwen [50, 51], and Parnas [45], have extended and refined the original SCR method and strengthened its formal foundations.

In 1989, Faulk [12] provided formal definitions for parts of the A-7 model. In particular, condition tables, a special class of tables in SCR requirements documents, are described as total functions and mode classes as finite state machines defined over events. Using mode classes to partition the state space is a form of abstraction that not only makes analysis of the specifications more efficient, it also reduces redundancy and makes the specifications easier to understand. A deficiency in the original A-7 requirements document, however, is that a mode class may be undefined in certain states; for example, if no weapon is allocated, the Weapons mode class is undefined. Faulk's model defines a mode class in every state; for example, when no weapon is allocated, the Weapons mode class is in mode `None`.

Using the original A-7 requirements document as a model, van Schouwen in 1990 [50] published a system-level requirements specification for the Water Level Monitoring System (WLMS), part of the shutdown system for a nuclear power plant. The WLMS specification extends the SCR method from software requirements to system requirements and demonstrates the use of the method to describe a system's accuracy and timing requirements as well as its functional requirements. The Four Variable Model [45] of Parnas and Madey provides a formal framework for the SCR method.

Four Variable Model. The Four Variable Model, which is illustrated in Figure 1, describes the required system behavior, including the required timing and accuracy, as a set of mathematical relations on four sets of variables—monitored and controlled variables and input and output data items. A *monitored variable* represents an environmental quantity that influences system behavior, a *controlled variable* an environmental quantity that the system controls. A black box specification of required behavior is given as two relations, REQ and NAT, from the monitored to the controlled quantities. NAT, which defines the set of possible values, describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment. REQ defines additional constraints on the system to be built as relations the system

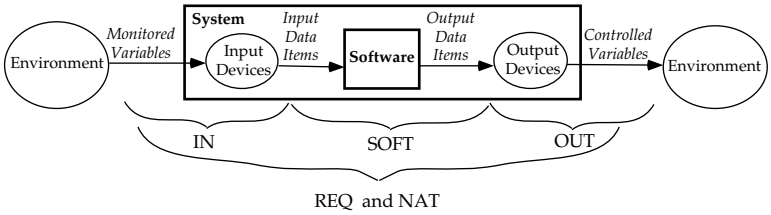

Figure 1: Four Variable Model.

must maintain between the monitored and the controlled quantities.

In the Four Variable Model, input devices (e.g., sensors) measure the monitored quantities and output devices set the controlled quantities, and *input* and *output data items* represent the values that the devices read and write. The relation IN defines the mapping from the monitored quantities to the input data items. The relation OUT defines the mapping from the output data items (e.g., actuators) to the controlled quantities. The use of monitored and controlled quantities to define the required behavior (rather than input and output data items) keeps the specification in the problem domain and allows a simpler specification.

Like the Four Variable Model, our requirements model can be used to describe both system requirements and software requirements. Our model defines the system requirements by describing REQ, the required relation between the monitored and controlled variables, and the software requirements by describing SOFT, the required relation between the input and output data items. Below, the term *input variable* (*output variable*) represents a monitored variable (a controlled variable) when REQ is being defined and an input data item (an output data item) when SOFT is being defined.

The next section reviews the constructs and tabular notation used in SCR requirements specifications in terms of the Four Variable Model. Because our initial requirements model emphasizes the system's functions, the discussion focuses on aspects of the Four Variable Model that describe functional behavior.

SCR Constructs. To specify the relations of the Four Variable Model in a practical and concise manner, four other constructs, each introduced in the A-7 requirements document [26], are useful. These are modes, terms, conditions, and events. A *mode class* is a state machine, defined on the monitored variables, whose states are called *system modes* (or simply *modes*) and whose transitions are triggered by events. Complex systems are defined by several mode classes operating in parallel. Mode classes reduce redundancy in the specifications by assigning a name (i.e., a mode) to a logical expression used many times in the specifications and using the name rather than repeating the logical expression. A *term* is an auxiliary function defined on input variables, modes, or other terms

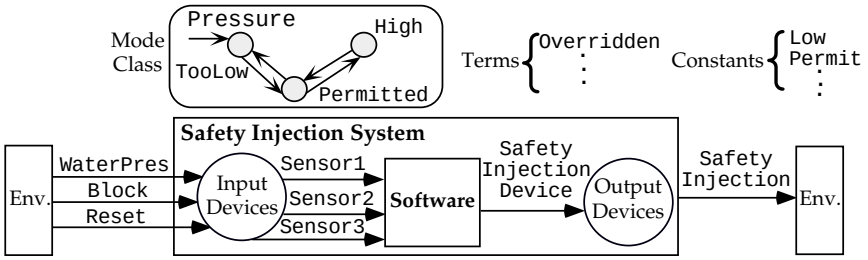

Figure 2: Requirements Specification for Safety Injection.

that helps make the specification concise. A *condition* is a predicate defined on one or more system entities (a system *entity* is an input or output variable, mode, or term) at some point in time. An *event* occurs when any system entity changes value. A special event, called an *input event*, occurs when an input variable changes value. A *conditioned event* occurs if an event occurs when a specified condition is true.

To illustrate the SCR constructs, we consider a simplified version of the control system for safety injection described in [8]. The system uses three sensors to monitor water pressure and adds coolant to the reactor core when the pressure falls below some threshold. The system operator blocks safety injection by turning on a "Block" switch and resets the system after blockage by turning on a "Reset" switch. Figure 2 shows how SCR constructs are used to specify the requirements of the control system within the framework of the Four Variable Model. Water pressure and the "Block" and "Reset" switches are represented as monitored variables, `WaterPres`, `Block`, and `Reset`; safety injection as a controlled variable, `Safety Injection`; each sensor as an input data item; and the hardware interface between the control system software and the safety injection system as an output data item.[1]

The specification for the control system includes a mode class `Pressure`, a term `Overridden`, and several conditions and events. The mode class `Pressure`, an abstract model of the monitored variable `WaterPres`, contains three modes, `TooLow`, `Permitted`, and `High`. At any given time, the system must be in one of these modes. A drop in water pressure below a constant `Low` causes the system to enter mode `TooLow`; an increase in pressure above a larger constant `Permit` causes the system to enter mode `High`. The term `Overridden` describes situations in which safety injection is blocked. An example of a condition in the specification is "`WaterPres` < `Low`". Events are denoted by the notation "@T". Two examples of events are the input event @T(`Block`=`On`) (the operator turns `Block` from `Off` to `On`) and the conditioned event @T(`Block`=`On`) WHEN `WaterPres` < `Low` (the operator turns `Block` to `On` when water pressure is below `Low`).

[1] The example omits the SCR bracketing notation, e.g., *...* for a mode, /.../ for an input data item, !...! for a term, $...$ for an enumerated value, etc.

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.

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

Table 2: Event Table for Overridden.

Mode	Conditions	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
Safety Injection	Off	On

Table 3: Condition Table for Safety Injection.

SCR Tables. The tabular notation used in SCR specifications facilitates industrial application of the method. Not only do engineers find tabular specifications of requirements easy to understand and to develop; in addition, tables can describe large quantities of requirements information concisely. Among the tables in SCR specifications are condition tables, event tables, and mode transition tables. Each table defines a mathematical function. A condition table describes an output variable or term as a function of a mode and a *condition*; an event table describes an output variable or term as a function of a mode and an *event*. A mode transition table describes a mode as a function of another mode and an event.

Tables 1–3 are part of REQ, the system requirements specification for the control system. Table 1 is a mode transition table describing the mode class Pressure as a function of the current mode and the monitored variable WaterPres. The table defines all events that change the value of the mode class Pressure. For example, the first row of Table 1 states, "If Pressure is TooLow and WaterPres rises to Low, then Pressure changes to Permitted." Events that do not change the value of the mode class are omitted from the table. For example, if Pressure is TooLow and WaterPres changes but does not reach Low, then Pressure is still TooLow after the event.

Table 2 is an event table describing the term Overridden as a function of

Pressure, 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 and Block becomes On when Reset is Off, then Overridden becomes *true*." In contrast, if the mode class is High and either Block or Reset changes (from Off to On or vice versa), then the value of Overridden does not change because the first row does not specify events involving either Block or Reset. The entry "False" in an event table means that no event can cause the variable defined by the table to assume the value in the same column as the entry; thus, the entry "False" in row 1 of Table 2 means that when the mode class is High, no event can cause Overridden to become *true*. The notation "@T(Inmode)" in a row of an event table describes system entry into the group of modes in that row; for example, "@T(Inmode)" in the second row of Table 2 means, "If the system enters TooLow or Permitted, then Overridden becomes *false*."

Table 3 is a condition table describing the controlled variable Safety Injection as a function of Pressure and 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.

While condition tables define total functions, event tables and mode transition tables may define partial functions. This is partly because some events cannot occur when certain conditions are true. For example, in the control system introduced above, the event @T(Pressure=High) WHEN Pressure=TooLow cannot occur, because starting from TooLow, the system can only enter Permitted when a state transition occurs. In other cases and as illustrated by the examples above, an event may occur that does not change the value of a variable defined by an event table or a mode transition table. In our formal requirements model (see below), we make the functions defined by event tables and mode transition tables total by assigning a variable its old value whenever the table does not explicitly define the variable's value.

3 Formal Requirements Model

Although earlier requirements models, namely, Faulk's automaton model [12], the model underlying van Schouwen's specification [50, 51], and the Four-Variable Model, define some aspects of the SCR requirements method, these models are too abstract to provide a formal basis for our tools. To provide a precise and detailed semantics for the SCR method, our model represents the system to be built as a finite state automaton and describes the input and output variables, conditions, events, and other constructs that make up an SCR specification in terms of that automaton. Our automaton model, a special case of the Four Variable Model, describes all monitored and controlled quantities, even those

which are naturally continuous, as discrete variables. Moreover, because our model abstracts away timing and imprecision, it describes the "ideal" system behavior. The system requirements are easier to specify and to reason about if the ideal behavior is defined first. Then, the required precision and timing can be specified separately. Reference [23] describes how our model can be extended to include continuous variables and to describe timing and accuracy requirements.

Although SCR requirements specifications may be nondeterministic, our initial model is formulated in terms of functions and is therefore restricted to deterministic systems. In some cases, nondeterminism may not be an error—in fact, requiring determinism can lead to overspecification of the requirements. However, like many practitioners and some researchers, we recognize and stress the advantages of deterministic specifications. As Berry has observed [3], "The importance of determinism cannot be overestimated; deterministic systems are one order of magnitude simpler to specify, debug, and analyze than nondeterministic ones."

Our requirements model, inspired by the formal security model presented in [34], defines sets of modes, entity names, values, and data types and a special function TY, which maps an entity to its legal values. The model defines system state in terms of the entities, a condition as a predicate on the system state, and an input event as a change in an input variable which triggers a new system state. It also describes how a set of functions, called table functions, can be derived from the SCR tables. These table functions define the transform T, a special case of REQ (or SOFT), which maps the current state and an input event to a new state. We present below excerpts from our requirements model [24] along with examples taken from the system requirements specification for the simple control system introduced above. To clarify the presentation, some definitions of the requirements model (e.g., the definitions of conditions and events) have been simplified.

System State. We assume the existence of the following sets.

- MS is the union of N nonempty, pairwise disjoint sets, namely, $M_1, M_2, \ldots, M_N$, called *mode classes*. Each member of a mode class is called a *mode*.
- TS is a union of data types, where each type is a nonempty set of values.
- $VS = MS \cup TS$ is the set of entity values.
- RF is a set of entity names r. RF is partitioned into four subsets: MR, the set of mode class names; IR, the set of input variable names; GR, the set of term names; and OR, the set of output variable names. For all $r \in RF$, $TY(r) \subseteq VS$ is the type (i.e., the set of possible values) of the entity named r. For all $r \in MR$, there exists i such that $TY(r) = M_i$; we say that r is the *mode class name* corresponding to M_i.

A *system state* s is a function that maps each entity name r in RF to a value. More precisely, for all $r \in RF$: $s(r) = v$, where $v \in TY(r)$. Thus, by assumption,

in any state s, the system is in exactly one mode from each mode class, and each entity has a unique value.

Example. In the sample system, the set of entity names RF is defined by

$$RF = \{\texttt{Block, Reset, WaterPres, Pressure, SafetyInjection, Overridden}\}.$$

The type definitions include

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

Conditions. Conditions are defined on the values of entities in RF. A *simple condition* is *true*, *false*, or a logical statement $r \odot v$, where $r \in RF$ is an entity name, $\odot \in \{=, \neq, >, <, \geq, \leq\}$ is a relational operator, and $v \in TY(r)$ is a constant value.[2] A *condition* is a logical statement composed of simple conditions connected in the standard way by the logical connectives $\wedge$, $\vee$, and $\neg$.

Events. The "@T" notation denotes various events. A *primitive event* is denoted @T$(r = v)$, where r is an entity in RF and $v \in TY(r)$. An *input event* is a primitive event @T$(r = v)$, where $r \in IR$ is an input variable. A *basic event* is denoted @T(c), where c is any simple condition. A *simple conditioned event* is denoted @T(c) WHEN d, where @T(c) is a basic event and d is a simple condition or a conjunction of simple conditions. Any basic event @T(c) can be expressed as the simple conditioned event @T(c) WHEN *true*. A *conditioned event* e is composed of simple conditioned events connected by the logical connectors $\wedge$ and $\vee$.

A simple conditioned event represents the logical expression defined by

$$\text{@T}(c) \text{ WHEN } d \;=\; \neg c \;\wedge\; c' \;\wedge\; d, \tag{1}$$

where the unprimed version of condition c denotes c in the old state and the primed version denotes c in the new state. Given $c = r \odot v$, we define c' as $c' = (r \odot v)' = r' \odot v$. Based on these definitions and the standard predicate calculus, any conditioned event can be expressed as a logical statement. An event occurs if the logical statement that the event represents evaluates to true for a given old state and a given new state.

Example. 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.

System (Software System). A *system* (*software system*) Σ is a 4-tuple, $\Sigma = (E^m, S, s_0, T)$, where

[2]Here, v is defined as a constant to keep the notation simple. Our formal model generalizes this definition, so that v may be any function defined on entities whose range is $TY(r)$.

- E^m is a set of input events.
- S is the set of possible system states.
- s_0 is a special state called the initial state.
- T, the system transform, is a partial function[3] from $E^m \times S$ into S.

A basic assumption, called the One Input Assumption, is that exactly one input event occurs at each state transition.[4]

Ordering the Entities. To compute an entity's value in the new state, the transform function may use the values of entities in both the old state and the new state. To describe the entities needed in the new state, each entity r is associated with a subset of RF called the *new state dependencies* set. For each input variable, the new state dependencies set is empty. For each entity defined by a condition table, the set contains the entity naming the associated mode class and all entities appearing in conditions in the table's body. For each entity defined by an event table or a mode transition table, the set contains all entities appearing in the table as part of a basic event and, if @T(Inmode) appears in the table, the associated mode class. These new state dependencies impose a partial ordering on the set RF. Using these sets, the entities in RF can be ordered as a sequence R, where for all i and j such that r_i and r_j belong to R, r_i depends directly on r_j implies that r_i follows r_j in R (that is, $i > j$).

Example. The condition table in Table 3 shows that the controlled variable `SafetyInjection` depends on two entities in the new state, the mode class `Pressure` and the term `Overridden`. Hence, the new state dependencies set for `SafetyInjection` is the set {`Pressure`, `Overridden`}. The partial ordering of the entities based on the new state dependencies is determined as follows: The three monitored variables are first because they only depend on changes in the environment. Next is the mode class `Pressure`, which depends on `WaterPres`. Next is the term `Overridden`, which depends on `Pressure` and two monitored variables, `Block` and `Reset`. The last entity in the partial ordering is `SafetyInjection`. A sequence R satisfying this partial ordering is

$$R = <\texttt{WaterPres, Block, Reset, Pressure, Overridden, SafetyInjection}>.$$

In sequence R, $r_1 =$ `WaterPres`, $r_2 =$ `Block`, $\cdots$, and $r_6 =$ `SafetyInjection`.

Table Functions. Each SCR table describes a *table function*, called F_i, which defines an output variable, a term, or a mode class r_i. Each entity r_i defined by a table is associated with exactly one mode class, M_j, $1 \leq j \leq N$. To represent

[3] T is a partial function because not all input events can occur in a given state. For example, in the control system, `Block` cannot change to `On` if `Block` is already `On`.

[4] Reference [2] presents an alternate definition of a conditioned event, namely, @T(c) WHEN $d = \neg c \wedge c' \wedge d \wedge d'$. When c and d define independent input variables, definition (1) and the One Input Assumption imply this alternate definition. Although we prefer definition (1) because the alternate definition makes the expression of certain conditioned events overly complex, we see advantages in making two WHEN operators available to specifiers, namely, WHEN $d \triangleq d$ and WHEN$'$ $d \triangleq d \wedge d'$.

the relation between an entity and a mode class, we define a function μ, where $\mu(i) = j$ if and only if entity r_i is associated with mode class M_j. Using this notation, $M_{\mu(i)}$ denotes the mode class associated with entity r_i.

Example. The single mode class in this specification is `Pressure`. Hence, $N = 1$, and $M_1 = \text{TY}(\texttt{Pressure}) = \{\texttt{TooLow, Permitted, High}\}$. Because all three entities defined by tables, namely, $r_4 =$ `Pressure`, $r_5 =$ `Overridden`, and $r_6 =$ `SafetyInjection`, are functions of `Pressure`, we have $\mu(4) = \mu(5) = \mu(6) = 1$.

Presented below for condition, event, and mode transition tables is a typical format and a description of how the table function is derived from a given table.

Modes	**Conditions**			
m_1	$c_{1,1}$	$c_{1,2}$	$\ldots$	$c_{1,p}$
m_2	$c_{2,1}$	$c_{2,2}$	$\ldots$	$c_{2,p}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
m_n	$c_{n,1}$	$c_{n,2}$	$\ldots$	$c_{n,p}$
r_i	v_1	v_2	$\ldots$	v_p

Table 4: Condition Table—Typical Format.

Condition Tables. Table 4 shows a typical format for a condition table with $n+1$ rows and $p+1$ columns. Each condition table describes an output variable or term r_i as a relation ρ_i defined on modes, conditions, and values. More precisely, $\rho_i = \{(m_j, c_{j,k}, v_k) \in M_{\mu(i)} \times C_i \times TY(r_i)\}$, where C_i is a set of conditions defined on entities in RF. The relation ρ_i must satisfy the following four properties:

1. The m_j and the v_k are unique.
2. $\cup_{j=1}^{n} m_j = M_{\mu(i)}$ (All modes in the associated mode class are included).
3. For all j: $\vee_{k=1}^{p} c_{j,k} = \mathit{true}$ (Coverage: The disjunction of the conditions in each row of the table is *true*).
4. For all $j, k, l,\ k \neq l$: $c_{j,k} \wedge c_{j,l} = \mathit{false}$ (Disjointness: The pairwise conjunction of conditions in a row of the table is always *false*).

To make explicit entity r_i's dependencies on other entities, we consider an alternate form F_i of the relation ρ_i. The four properties above ensure that F_i is a total function. Properties 1 and 4 ensure that F_i is a function, while Properties 2 and 3 guarantee totality. To define F_i, we require the new state dependencies set, $\{y_{i,1}, y_{i,2}, \ldots, y_{i,n_i}\}$, where $y_{i,1}$ is the entity name for the associated mode class and $y_{i,2}, \ldots, y_{i,n_i}$ are entities that appear in some condition c in C_i. Based on this set and ρ_i, we define r_i as a function F_i as follows:

$$r_i = F_i(y_{i,1}, \ldots, y_{i,n_i}) = \begin{cases} v_1 & \text{if } \vee_{j=1}^{n} (y_{i,1} = m_j \wedge c_{j,1}) \\ v_2 & \text{if } \vee_{j=1}^{n} (y_{i,1} = m_j \wedge c_{j,2}) \\ \vdots & \\ v_p & \text{if } \vee_{j=1}^{n} (y_{i,1} = m_j \wedge c_{j,p}). \end{cases}$$

The function F_i is called a *condition table* function.

Example. Based on the new state dependencies set {`Pressure, Overridden`} and Table 3, the condition table function for `Safety Injection`, denoted F_6, is defined by

`SafetyInjection` =

$$F_6(\texttt{Pressure}, \texttt{Overridden}) = \begin{cases} \texttt{Off} & \text{if} \quad \texttt{Pressure} = \texttt{High} \vee \texttt{Pressure} = \texttt{Permitted} \vee \\ & \qquad (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \mathit{true}) \\ \texttt{On} & \text{if} \quad \texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \mathit{false} \end{cases}$$

Modes	**Events**			
m_1	$e_{1,1}$	$e_{1,2}$	$\ldots$	$e_{1,p}$
m_2	$e_{2,1}$	$e_{2,2}$	$\ldots$	$e_{2,p}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
m_n	$e_{n,1}$	$e_{n,2}$	$\ldots$	$e_{n,p}$
r_i	v_1	v_2	$\ldots$	v_p

Table 5: Event Table—Typical Format.

Event Tables. Table 5 illustrates a typical format for an event table with $n+1$ rows and $p+1$ columns. Each event table describes an output variable or term r_i as a relation ρ_i between modes, conditioned events, and values, i.e., $\rho_i = \{(m_j, e_{j,k}, v_k) \in M_{\mu(i)} \times E_i \times TY(r_i)\}$, where E_i is a set of conditioned events defined on entities in RF. The relation ρ_i must satisfy the following two properties:

1. The m_j and the v_k are unique.
2. For all $j, k, l,\ k \neq l$: $e_{j,k} \wedge e_{j,l} = \mathit{false}$ (Disjointness: The pairwise conjunction of events in a row of the table is always *false*).

As with condition tables, we make explicit r_i's dependency on other entities by extending the relation ρ_i to an alternate form F_i. The One Input Assumption and the two properties above ensure that F_i is a function. The "no change" part of F_i's definition (see below) guarantees totality.

To define F_i, we require both the new state dependencies set $\{y_{i,1}, \ldots, y_{i,n_i}\}$ and an *old state dependencies* set $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}\}$, which contains the associated mode class $x_{i,1}$ along with all entities appearing in some $e_{i,j}$ in the associated table. Based on the old state dependencies, the new state dependen-

Old Mode	Event	New Mode
m_1	$e_{1,1}$	$m_{1,1}$
	$e_{1,2}$	$m_{1,2}$
	$\cdots$	$\cdots$
	e_{1,k_1}	m_{1,k_1}
$\cdots$	$\cdots$	$\cdots$
m_n	$e_{n,1}$	$m_{n,1}$
	$e_{n,2}$	$m_{n,2}$
	$\cdots$	$\cdots$
	e_{n,k_n}	m_{n,k_n}

Table 6: Mode Transition Table—Typical Format.

cies, and ρ_i, F_i is defined by

$$r_i' = F_i(x_{i,1}, \ldots, x_{i,m_i}, y_{i,1}', \ldots, y_{i,n_i}') = \begin{cases} v_1 & \text{if } \vee_{j=1}^{n} (x_{i,1} = m_j \wedge e_{j,1}) \\ v_2 & \text{if } \vee_{j=1}^{n} (x_{i,1} = m_j \wedge e_{j,2}) \\ \vdots & \\ v_p & \text{if } \vee_{j=1}^{n} (x_{i,1} = m_j \wedge e_{j,p}) \\ r_i & \text{otherwise (i.e., no change).} \end{cases}$$

The function F_i is called an *event table* function. Note that if none of the $e_{j,k}$ occurs, then the entity r_i defined by F_i retains its value in the old state.

Example. Both the old state dependencies set and the new state dependencies set for `Overridden`, {`Block, Reset, Pressure, Overridden`}, and {`Block, Reset, Pressure`}, can be derived from Table 2. Based on these sets and Table 2, the event table function for `Overridden` is defined by

$\texttt{Overridden}' =$
$F_5(\texttt{Block}, \texttt{Reset}, \texttt{Pressure}, \texttt{Overridden}, \texttt{Block}', \texttt{Reset}', \texttt{Pressure}') =$

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

Mode Transition Tables. Table 6 shows a typical format for a mode transition table for an entity r_i that names a mode class $M_{\mu(i)}$. The table describes r_i as a relation $\rho_i = \{(m_j, e_{j,k}, m_{j,k}) \in M_{\mu(i)} \times E_i \times M_{\mu(i)}\}$, where E_i is a set of conditioned events defined on entities in RF. The relation ρ_i must satisfy the following four properties:

1. The m_j are unique.
2. For all $k \neq l$, $m_{j,k} \neq m_{j,l}$, and for all j and for all k, $m_j \neq m_{j,k}$.
3. For all $j, k, l,\ k \neq l$: $e_{j,k} \wedge e_{j,l} = \mathit{false}$ (Disjointness: The pairwise conjunction of conditioned events in a row of the table is always false).
4. Let m_0 be the initial mode. Then, $M_{\mu(i)} \subseteq \{m \mid Q^*(m_0, m)\}$, where $Q(m_1, m_2)$ if and only if $\rho_i(m_1, e, m_2)$ for some e and Q^* is the reflexive and transitive closure of Q (Reachability: Each mode must be reachable from the initial mode).

It is easy to show that a mode transition table with the format in Table 6 can be expressed in the format shown in Table 5 for an event table. Hence, a mode transition table can be expressed as an event table function.

Example. Based on Table 1, the old and new dependencies sets for the mode class `Pressure` are {`WaterPres, Pressure`} and {`WaterPres`}. Given these sets and Table 1, the table function for `Pressure` is defined by

$\texttt{Pressure}' =$
$F_4(\texttt{Pressure}, \texttt{WaterPres}, \texttt{WaterPres}') =$

$$\begin{cases} \texttt{TooLow} & \text{if } \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres}' < \texttt{Low} \wedge \texttt{WaterPres} \not< \texttt{Low} \\ \texttt{High} & \text{if } \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres}' \geq \texttt{Permit} \wedge \texttt{WaterPres} \not\geq \texttt{Permit} \\ \texttt{Permitted} & \text{if } (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{WaterPres}' \geq \texttt{Low} \wedge \texttt{WaterPres} \not\geq \texttt{Low}) \vee \\ & \quad (\texttt{Pressure} = \texttt{High} \wedge \texttt{WaterPres}' < \texttt{Permit} \wedge \texttt{WaterPres} \not< \texttt{Permit}) \\ \texttt{Pressure} & \text{otherwise.} \end{cases}$$

4 Automated Consistency Checking

Listed below are consistency checks derived from our formal requirements model. These checks, which determine whether the specifications are well-formed, are independent of a particular system state. Because they can be performed without executing the system (or the specification), these checks are a form of static analysis.

- **Proper Syntax.** Each component of the specification has proper syntax. For example, each condition and event is well-formed.
- **Type Correctness.** Each variable has a defined type, and all type definitions are satisfied. For example, given any expression of the form $r = v$, where r is an entity and v is a value, $v \in TY(r)$ must hold.
- **Completeness of Variable and Mode Class Definitions.** The value of each controlled variable, term, and mode class is defined. (Most variables will be defined by tables, but standard mathematical definitions may be given for some controlled variables and terms.)

Mode	Conditions	
`High, Permitted`	True	False
`TooLow`	`Overridden`	`Overridden`
`Safety Injection`	`False`	`True`

Table 7: Modified Table for `Safety Injection`.

- **Initial Values.** Initial values are defined for all mode classes, input variables, terms, and output variables. Initial values are not required for entities defined by condition tables, since they can be derived from the tables.
- **Reachability.** Every mode in a mode class is reachable from the initial mode of the mode class. This is a check of Property 4 for Mode Transition Tables.
- **Disjointness.** To make the specifications deterministic, each condition, event, and mode transition table must satisfy the Disjointness property. That is, in a given state, each controlled variable, mode class, and term is defined uniquely.
- **Coverage.** Each condition table satisfies the Coverage property. That is, each variable described by a condition table is defined everywhere in its domain.
- **Lack of Circularity.** No circularities exist among the new dependencies sets. This property checks that the entities are partially ordered.

Clearly, some checks must precede others. For example, checks for proper syntax must precede type checking, and type checking should precede checking for the Coverage property.

Checking for Disjointness and Coverage. The most computationally expensive checks are checks for Disjointness and Coverage. To check these properties, the consistency checker determines whether a logical expression is a tautology. For example, to check two entries c_1 and c_2 in a row of a condition table for Disjointness, the consistency checker evaluates the logical expression $c_1 \wedge c_2 = \mathit{false}$. To check the entries c_1, c_2, ..., c_n in a row of a condition table for Coverage, the tool evaluates the logical expression $c_1 \vee c_2 \vee \ldots \vee c_n = \mathit{true}$. To determine whether these logical expressions are tautologies, our tool applies a tableaux-based decision procedure that encodes the algorithm in [48].

Examples. Checking the consistency of Table 7, a modification of the condition table in Table 3, reveals four errors. The third row has two type errors: `Safety Injection` has the values `Off` and `On`, not `False` and `True`. The second

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

Table 8: Modified Table for Overridden.

row violates two properties of condition tables, namely, Coverage (Overridden $\vee$ Overridden = Overridden $\neq$ *true*) and Disjointness (Overridden $\wedge$ Overridden = Overridden $\neq$ *false*).

Disjointness, the second property required of event tables, is violated if events in two different columns, say e_1 and e_2, overlap, i.e., $e_1 \wedge e_2 \neq$ *false*. Table 8 is a variation of the event table in Table 2. Running the consistency checker detects a Disjointness error in the second row of Table 8. In checking for Disjointness, the consistency checker evaluates the expression, [@T(Block=On) WHEN Reset=Off] $\wedge$ [@T(Block=On) $\vee$ @T(Reset=On)] = *false*. Below, we show that this expression is not a tautology.

[@T(Block = On) WHEN Reset = Off] $\wedge$ [@T(Block = On) $\vee$ @T(Reset = On)]

= [@T(Block = On) WHEN Reset = Off $\wedge$ @T(Block = On)] $\vee$
[@T(Block = On) WHEN Reset = Off $\wedge$ @T(Reset = On)] (Distributive Law)

= [Block = Off $\wedge$ Block$'$ = On $\wedge$ Reset = Off $\wedge$ Block = Off $\wedge$ Block$'$ = On] $\vee$
[Block = Off $\wedge$ Block$'$ = On $\wedge$ Reset = Off $\wedge$ Reset = Off $\wedge$ Reset$'$ = On] (By Def. (1))

= [Block = Off $\wedge$ Block$'$ = On $\wedge$ Reset = Off] $\vee$ *false* (One Input Assumption)

= Block = Off $\wedge$ Block$'$ = On $\wedge$ Reset = Off

$\neq$ *false*

Because the expression does not evaluate to *false*, the specified behavior is nondeterministic, i.e., there is at least one pair of states (s, s'), where the event expression evaluates to *true*. In particular, if in TooLow or Permitted mode the operator turns Block on when Reset is off, the system may nondeterministically change Overridden to *true* or to *false*.

Some checks, such as syntax and type checking, are straightforward. More complex are checks that depend on definitions, other than type definitions, in different parts of the specification or checks that require deductive reasoning. Consider, for example, checking the mode table in Table 1 for nondeterminism. Nondeterminism can occur only if events in the second and third rows overlap. These rows overlap if the expression @T(WaterPres$\geq$Permit) $\wedge$ @T(WaterPres<Low) evaluates to *true* in at least one situation. Based on (1), this expression can be rewritten as WaterPres$' \geq$ Permit $\wedge$ WaterPres $\ngeq$ Permit $\wedge$ WaterPres$' <$ Low $\wedge$ WaterPres $\nless$ Low. By assumptions on the constants, Permit $>$ Low. This assumption and WaterPres$' \geq$ Permit imply

WaterPres′ $\geq$ Low, a contradiction. Hence, the expression is always *false* and the defined behavior deterministic.

We have provided a semantic framework to reason formally about assumptions, such as Permit $>$ Low, that underlie a specification. Because, in general, mechanical evaluation of such expressions is undecidable, we are devising algorithms to identify and evaluate decidable subsets of these expressions under a set of assumptions (see [4] for details). For the general case, the tool may need user feedback to complete certain checks.

Efficiency of Our Technique. The analysis performed by our consistency checker is quite efficient because it is based on static checks of components of an SCR requirements specification rather than some form of reachability analysis. As noted above, some checks, such as checks for proper syntax, type correctness, and the completeness of definitions, are easy. Moreover, mode reachability can be computed using standard search techniques in time linear in the number of transitions in the mode class. Similarly, checking for lack of circularity requires time linear in the number of arcs in the new state dependencies graph.

Our definition of the transform function T simplifies checking that T is *complete* (for each input event that may occur, at least one new system state is completely defined) and *deterministic* (at most one new system state is defined). Determinism and completeness are guaranteed if the following properties are satisfied for each input event that occurs (see [24] for details):

- Lack of circularity
- Properties 1–4 for condition tables
- Properties 1–2 for event tables
- Properties 1–3 for mode transition tables

As we indicate above, checking for Disjointness and Coverage amounts to checking that logical formulas on conditions or events are tautologies. Although tautology checking may have worst case behavior that is exponential in the size of the expression [16], we expect this not to occur in practice. In particular, the use of modes to partition the system state means that Disjointness and Coverage checking is decomposed into small, independent subproblems. Thus Coverage checking for condition tables can be performed independently for each row (rather than checking all rows together for missing cases). Similarly, each condition table, event table, and mode transition table can be checked for Disjointness by analyzing each pair of cells in a row (rather than checking two columns at a time). Another benefit of using modes arises in analyzing Coverage errors: In our consistency checker, the same analysis that detects an error also identifies the specific cause of the error (i.e., the missing cases) and its precise location in the table. Section 6 compares our approach to Disjointness and Coverage checking with other approaches.

Our experience with consistency checking is that the number of subproblems and the size of each subproblem grows rather slowly. In contrast, using state reachability techniques, such as model checking, to check for Disjointness and Coverage would be more expensive, because the cost of reachability analysis increases exponentially with the size of the specification. Thus we expect that our techniques would not suffer the state explosion problem that plagues techniques such as model checking. Section 5 demonstrates that our technique for consistency checking scales to industrial-strength systems. Because consistency checking is cheap, it makes sense to perform consistency checking early and to postpone more expensive checks, such as model checking, until later on.

Prototype Consistency Checker. A prototype consistency checker that performs all of the above checks has been implemented [20]. The tool is coded in C++ and runs on X-Windows with Motif widgets to support its user interface. In a typical session with the consistency checker, the user edits a specification and then runs the consistency checker to test for selected properties. The tool runs the selected checks, listing any errors it finds. The user may select one of the listed errors to display the parts of the specification that produced the error (e.g., the specific rows or entries of the relevant table). In the case of a Coverage or a Disjointness error, the tool also displays a counterexample. Once the user has made corrections, he or she can rerun the consistency checker to ensure that the error has been properly corrected and no new errors have been introduced.

5 Applying Consistency Checks

To evaluate the utility of checking requirements specifications for consistency, we conducted two experiments. In the experiments, our consistency checker was used to analyze both the condition tables and the mode transition tables in a revised version [1] of the requirements specification for the A-7's Operational Flight Program (OFP). The OFP contains more than 16K lines of tight assembly code. The revised version corrects several errors in the original specification [26] and uses a new tabular format to specify mode transitions [12, 39].

The results of these two experiments demonstrate the efficiency of our techniques. Running on a Sun SPARCstation 20, our consistency checker checked all of the condition tables in the A-7 requirements specification for both Disjointness and Coverage and all of the mode transition tables in the specification for Disjointness. The tool required a total computation time of 245 seconds to perform the entire analysis. This time includes a time of 45 seconds to check the syntax and type correctness of the tables and only 200 seconds for all of the Disjointness and Coverage checks, 30 seconds for the condition tables and 170 seconds for the mode transition tables.

Below, we briefly describe the two experiments. Because each mode transition table is much larger than any of the condition tables and thus more complex

Class of Error	No. of Occurrences	Explanation
Slewing Variable	9	Behavior for 3rd value of variable `Slewing` is missing.
GRTest	4	Some tables do not specify behavior for all `GRTest` submodes.
Steering Phase	3	Early document used 3 values to describe steering phases. Revised document uses 4 values, but some tables have not been updated.
Application-Specific	1	(`OTS` $\vee$ `Range to RMax` < 0) and NOT (`range to target` ≤ 10 mi.) do not cover the domain.

Table 9: Errors Detected in the Condition Tables in [1].

to analyze, we describe the analysis of the mode transition tables in somewhat more detail.

Checks on Condition Tables. In the first experiment, our tool checked 36 condition tables, a total of 98 rows, for both Coverage and Disjointness. The tool found 19 errors. Seventeen of these, distributed over 11 tables, proved to be legitimate errors. Determining that the remaining two "errors" were correct required knowledge that our simple tool lacked. Both cases involved three values describing combined settings of two input devices. We confirmed by hand that the two rows containing these values satisfy the two properties.

Table 9 describes the detected errors. Interestingly, all 17 are Coverage errors. In this example, the reason why inspection is more likely to detect Disjointness errors than Coverage errors is that all the information needed to detect Disjointness errors is in the table, whereas the information needed to detect Coverage errors is not. Finding Coverage errors requires knowledge of all values a variable can take on. In [1], some type information is provided, not in the table, but elsewhere in the requirements document; in several cases, it is omitted entirely. (Omission of type information is a consequence of the semiformal nature of the A-7 requirements document. Using our tool, which enforces the type definitions of our requirements model, helps eliminate such errors. The tool checks that each entity in the specification has a corresponding type definition and that each logical expression in the specification satisfies the type definitions.)

Checks on Mode Transition Tables. In a second experiment, our tool checked all three mode transition tables in the A-7 requirements specification for Disjointness.[5] The three mode classes contain a total of 48 different modes (18 modes in the first mode class, 7 modes in the second, and 23 modes in the

[5]The authors learned recently of an experiment, reported in 1983, which also detected

third). The mode transition tables together contain a total of 700 rows, each row a simple conditioned event of the form described below. In analyzing the tables, our tool found 57 Disjointness errors.[6] Although many of the 57 instances of Disjointness violations that the tool detected are undoubtedly errors, a few probably are not, since some detected events may be impossible. (As noted above, some events cannot occur when certain conditions are true.) Of the 57 errors, 54 occurred in the mode transition table for the Weapons mode class and the remaining 3 in the mode transition table for the Alignment, Navigation, and Test mode class. Many of the Disjointness errors in the table for the Weapons mode class were instances of nondeterminism present in the prose requirements documents from which the A-7 requirements specification [26, 1] was partially derived.

Table 10, an excerpt from the revised A-7 specification, shows a small part of the mode transition table for the Alignment, Navigation, and Test mode class. (In [1], the complete table is more than 14 pages long.) This table has the same formal definition as Table 1 but a more structured format. In contrast to Table 1, headings in the middle column are simple conditions defined on input variables and terms. Each row of the middle column of Table 10 must contain a simple conditioned event. Consecutive rows of Table 10 that are not separated by a dashed line (e.g., rows 2 through 5) represent the disjunction of the simple conditioned events in the rows. In Table 10, the notation "@T" ("@F") denotes the event occurring when the corresponding condition becomes true (false), "t" ("f") means that the corresponding condition is true (false), and "–" means that the corresponding condition may be either true or false. To clarify the relationship between the two formats for mode transition tables, Table 11 translates the first five rows of Table 10 into the alternate format. (Except for the asterisks denoting mode names, Tables 10 and 11 omit the special SCR brackets.)

Table 10 illustrates a Disjointness error detected by our consistency checker. Because the two conditioned events marked by arrows are mapped to different new modes, they should not overlap. That they do overlap is an error. Overlap in these two events allows the system to transfer nondeterministically from the mode Inertial (*I*) to either the mode Airborne Alignment (*Airaln*) or the mode Doppler Inertial (*DI*). An event that triggers either transition is

$$@\mathrm{T}(\texttt{Doppler up}) \text{ WHEN } \neg(\texttt{CA stage complete}) \wedge \texttt{IMSMODE} = \texttt{Gndal} \wedge$$
$$\texttt{latitude} > 70 \text{ deg} \wedge \texttt{latitude} \not> 80 \text{ deg} \wedge \neg(\texttt{present position entered}).$$

Disjointness errors in mode transition tables [39]. However, our algorithm is more general than the algorithm described in [39].

[6] Reference [21] describes a similar experiment in which we used an early version of the consistency checker and detected fewer errors. Although the early tool used the same algorithm, it only looked for one Disjointness error for every pair of disjunctions it analyzed. The current tool is designed to find *all* Disjointness errors and thus detected additional errors.

Current Mode	!present position entered!	/ACAIRB/=Yes	!Data 23!=Sea	!CA stage complete!	!CL stage complete!	!IMS Up!	!latitude! gt 70 deg.	!latitude! gt 80 deg.	!Doppler up!	!Doppler coupled!	/IMSMODE/=$Cndal$	/IMSMODE/=$Norm$	/IMSMODE/=$Iner$	/IMSMODE/=$Mag sl$	/IMSMODE/=$Grid$	!Air velocity test passed!	!Pitch small! AND !Roll small!	New Mode
I	-	@F	f	-	-	-	-	-	-	-	t	-	-	-	-	-	-	*Landaln*
	-	-	-	f	-	-	-	-	@T	-	t	-	-	-	-	-	-	*Airaln*
	-	-	-	f	-	-	-	-	@T	-	-	t	-	-	-	-	-	
	-	-	-	f	-	-	-	-	t	-	@T	-	-	-	-	-	-	
	-	-	-	f	-	-	-	-	t	-	-	@T	-	-	-	-	-	
	f	-	-	t	-	-	f	-	@T	-	t	-	-	-	-	-	-	*DIG*
	f	-	-	t	-	-	f	-	@T	-	-	t	-	-	-	-	-	
	-	-	-	t	-	-	f	-	t	-	@T	-	-	-	-	-	-	
	-	-	-	t	-	-	f	-	t	-	-	@T	-	-	-	-	-	
	f	-	-	-	-	-	t	f	@T	-	-	t	-	-	-	-	-	*DI*
	f	-	-	-	-	-	t	f	@T	-	t	-	-	-	-	-	-	
	-	-	-	t	-	-	t	f	t	-	@T	-	-	-	-	-	-	
	-	-	-	t	-	-	t	f	t	-	-	@T	-	-	-	-	-	
	f	-	-	-	-	-	-	f	@T	t	-	-	t	-	-	-	-	
	f	-	-	-	-	-	-	f	t	@T	-	-	t	-	-	-	-	
	@T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*OLB*
	-	-	-	-	-	-	-	-	-	-	-	-	-	@T	-	-	-	*Mag sl*
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	@T	-	-	*Grid*
	-	-	-	-	-	@F	-	-	-	-	-	-	-	-	-	-	-	*IMS fail*
	-	-	-	-	-	-	-	@T	-	-	-	-	-	-	-	-	-	*PolarI*

Table 10: Error Detected in the Mode Transition Table for the Alignment, Navigation, and Test Mode Class [1].

In its analysis, the consistency checker examined 4319 conjunctions. Each conjunction was of the form $e_1 \wedge e_2$, where e_1 and e_2 are simple conditioned events from two different rows of a mode transition table in the format shown in Table 10. The two rows have the same current mode but different new modes. The arrow in Table 10 indicates two such rows. In the analysis, the maximum number of @T, @F, t, or f entries in the analyzed conjunctions (that is, entries not marked "–") was 18. The average number of @T, @F, t, or f entries in the analyzed conjunctions was 5.7. Of the 4319 conjunctions, 1691 required analysis by the tautology checker, while the remainder were analyzed separately because they were trivial. Our experience is that the relatively small-sized logical expressions analyzed in our experiments do not require sophisticated algorithms, such as BDDs [6].

Current Mode	Event	New Mode
I	@F(ACAIRB = Yes) WHEN NOT(Data23 = Sea) ∧ IMSMODE = Gndal	*Landaln*
I	@T(Doppler up) WHEN NOT(CA stage complete) ∧ IMSMODE = Gndal ∨ @T(Doppler up) WHEN NOT(CA stage complete) ∧ IMSMODE = Norm ∨ @T(IMSMODE = Gndal) WHEN NOT(CA stage complete) ∧ Doppler up ∨ @T(IMSMODE = Norm) WHEN NOT(CA stage complete) ∧ Doppler up	*Airaln*
...	...	...

Table 11: Mode Transition Table in Table 10 Expressed in the Alternate Format.

Manual vs. Automated Checks. Prior to its publication, the revised A-7 requirements document was carefully reviewed by two teams, one made up of NRL computer scientists (including the third author), the other composed of engineers at the Naval Air Warfare Center who maintained the OFP. As noted above, our tools detected many significant errors that the reviewers missed.

That errors were detected should not diminish the credit due the reviewers, who did very well given the large volume and complexity of the requirements data. Tools, such as ours, can complement the efforts of software developers. Human effort is crucial to acquiring the requirements information and expressing it precisely. Further, after errors are detected in the specification, human intervention is needed to correct them. However, once the developers have a reasonable draft of the requirements specifications, software tools provide a quick, effective means of checking the specification for properties, such as those described in Section 4. Not only are tools more effective than people for checking these properties; in addition, they can significantly reduce a labor-intensive task that humans find tedious and boring.

Another important feature of our tool is its low cost. In the Darlington certification effort, which cost over $40M, reviewers checked the requirements specifications for application-independent properties, such as Disjointness and Coverage. In addition, they searched for discrepancies between the requirements specifications and the code specifications. A tool that compares the specifications with a refinement will be more complex than our consistency checker. However, this does not diminish the value of our tool. Parnas has observed that the "majority of the theorems that arose in the documentation and inspection of the Darlington Nuclear Plant Shutdown Systems" were simple properties and that the reviewers analyzed trivial tables for such properties in documents weighing 40 kilograms [41]. Using tools to do this analysis should cost far less than using people.

6 Related Work.

A number of industrial organizations, including Bell Laboratories [28], Grumman [39], Ontario Hydro [43], and the Software Productivity Consortium [14, 15], have adapted the SCR method to specify the requirements of practical systems. Additionally, Parnas [40] and Janicki [33] have developed formal semantics for many classes of tables. The condition tables and event tables defined formally in Section 3 are special cases of their Inverted Tables. For another class of tables defined by Parnas called Program Tables [42], a tool has been developed which checks for Disjointness and Coverage errors before proceeding with additional analyses [10]. Below, we briefly describe other related work: two other automated techniques for analyzing tabular specifications for consistency and three other approaches for detecting errors in SCR requirements specifications.

Decision Table Processors. Decision tables [38, 31], an early tabular notation, have been used for many years to specify software requirements because tables are easier to read than expressions in predicate logic. Typically, decision tables associate each system input with one or more actions. A general difference between SCR and methods based on decision tables is that, unlike SCR, which associates a separate table with each output, term, and mode class in the specification, methods based on decision tables do not usually decompose a specification into smaller, more manageable pieces. A notable exception is an early specification technique, called Systematics [17], which decomposes decision tables into smaller tables similar to SCR's condition tables.

Various techniques have been developed to process decision tables. Although most techniques focus on code generation, a few have been designed to do consistency checking. DETRAN (DEcision TRANslator) is an early example of a technique that checks decision tables for both Disjointness and Coverage. It can also translate the contents of a given table to either FORTRAN or COBOL [30].

Tablewise. Tablewise [29], a recent technique for processing decision tables, checks a table for both Disjointness (called "consistency") and Coverage (called "completeness"). Tablewise improves on earlier techniques based on decision tables in that it supports nonboolean variables. The format of the tables analyzed by Tablewise is similar to the format of Table 10, except conditions label rows (rather than columns) and each column is associated with an action. A significant difference between the tables processed by Tablewise and SCR tables is that the former do not use modes.

RSML. The Requirements State Machine Language (RSML) [32, 35, 19], which was developed to describe real-time process control systems, uses a combination of graphical and tabular notations. RSML's graphical notation (e.g., its superstates and AND-decomposition) is largely borrowed from Statecharts [18].

Additional features of RSML include directed one-to-one communication among high-level state machines and the introduction of AND/OR tables to describe the guards on state transitions. Each table in RSML describes the conditions under which a given state machine can make a transition from one state to a second state under a specific input event. An output event may be associated with each table. The format of RSML tables resembles the format of Table 10, except (as in Tablewise) conditions label the rows of the table rather than the columns. Also, in contrast to Table 10, different conditions need not be disjoint, events do not appear as table entries, and modes (that is, modes defined explicitly as functions of input variables) are not used.

A prototype tool has been developed for checking RSML specifications for "completeness" (i.e., every possible input event and the system's response to the event must be stated explicitly) and "consistency" (i.e., no input event can cause a transition to two different system states). The tool, which has been applied to large portions of the requirements specification of TCAS II, a collision avoidance system for commercial aircraft, detected errors not caught by an extensive manual review.

Consistency Checking in Tablewise, RSML, and SCR. RSML and SCR each support next state semantics that simplify checking for completeness and determinism. In either case, these checks may be performed by decomposing the problem into smaller subproblems and avoiding the state explosion inherent in reachability techniques. However, a comparison of these two requirements languages and of the analysis required to perform Disjointness and Coverage checks is made difficult by the major differences in the RSML and SCR semantics. Further investigation is required for a quantitative comparison of these two methods on realistic systems.

Because decision table processors, such as Tablewise, do not structure a specification using mode classes, they cannot exploit the reduced analysis for Coverage and Disjointness errors that mode classes make possible. In addition to reducing the analysis needed to check certain properties, our use of modes also makes it easy to determine the specific cause of an error. When Tablewise detects a Coverage error, additional analysis, called a "structural analysis," must be performed to determine the missing cases [29]. This additional analysis is unnecessary for our consistency checker: As noted above, the specific cases that have been omitted as well as the precise row of the table in which the missing case was detected are byproducts of the analysis our tool does in checking for Coverage errors.

Mechanical Proof Systems. Parnas describes ten small theorems that must be true of sample specifications expressed in his tabular notation (similar to other SCR notation) and challenges the developers of automated proof systems to prove the theorems [41]. Two of the theorems, the Domain Coverage Theorem and the Disjoint Domains Theorem, are slight variations of our Coverage and

Disjointness properties. SRI researchers accepted Parnas' challenge. In a recent paper [47], they describe the mechanical proof of nine of Parnas' theorems using the "tcc-strategy"(tcc's are type-correctness conditions) of SRI's proof system PVS [9]. That PVS can prove such theorems easily is not too surprising, since the proofs are based on very simple logic. What is noteworthy about the PVS experiment is that the theorems were proven automatically.

Model Checking. Atlee and Gannon have demonstrated the utility of model checking [7] for detecting application-dependent errors in SCR requirements specifications [2]. However, where our consistency checker tests all tables and other definitions (e.g., definitions of types, constants, etc.) in an SCR specification, their tool analyzes properties of the mode transition tables only.

Detecting Errors by Inspection. A recent experiment [46] compares the effectiveness of three different inspection methods for detecting errors in SCR requirements specifications. Many of the errors of interest in the experiment can be automatically detected by our consistency checker. Using a tool like ours in conjunction with inspection is likely to detect more errors than either alone.

7 Requirements Process

We envision the following process for developing requirements specifications. Although such a process is an idealization of a real-world process [44], it shows how tools such as our consistency checker can be used to produce high quality requirements specifications.

1. A formal notation, such as the SCR notation, is used to specify the requirements.

2. An automated consistency checker is used to check the specification for syntax and type correctness, coverage, determinism, and other application-independent properties.

3. The specification is executed symbolically using a simulator to ensure that it captures the customers' intent.

4. In the later stages of the requirements phase, mechanical support is used to analyze the specification for application properties. Initially, a small subset with fixed parameters and only a few states is extracted from the specification and a tool, such as a model checker, is used to detect violations of the properties. This may be repeated, each time with a different or larger subset. Once there is sufficient confidence in the specification, a mechanical proof system may be used to verify the complete requirements specification or, more likely, safety-critical components.[7]

[7] A similar approach that combines model checking and mechanical theorem proving is suggested in [36].

8 Concluding Remarks

Based on our experience with automated consistency checking to date, we have four conclusions:

- Tools for consistency checking can be highly effective for detecting errors in requirements specifications. Not only can such tools find errors people miss; they can liberate people from the tedious and error-prone task of checking specifications for consistency.
- Using properly designed tools for consistency checking is significantly cheaper than using people.
- Computer-based analysis requires an explicit formal semantics, such as that provided by our requirements model. This semantics provides the basis for algorithms that do the analysis.
- The formal method on which our tools are based scales up. The method detected a significant number of errors in a medium-size real-world specification. The structuring of the specifications using modes contributes significantly to the efficiency with which consistency checks, especially Disjointness and Coverage checks, can be performed on large requirements specifications.

In addition to the consistency checker described in this paper, our current toolset includes a specification editor and a simulator. We are also developing a verifier that checks SCR requirements specifications for application properties. An option being considered is to link the toolset with a mechanical proof system to support both automated consistency checking and computer-assisted verification. This would relieve us of the difficult and error-prone task of encoding our own proof engine.

We expect our requirements model to provide a solid foundation for a suite of analysis tools. We also expect the process outlined above, which uses formal notation to specify requirements and computer-supported formal analysis to detect errors, to produce high quality requirements specifications. Such specifications should produce systems that are more likely to perform as required and less likely to lead to accidents. They should also lead to significant reductions in software development costs.

Acknowledgments

We gratefully acknowledge the work of C. Gasarch, D. Kiskis, and A. Rose on the design and implementation of the consistency checker. P. Clements detected an error in our formal definition of the Reachability property. S. Faulk suggested

that we define our requirements model within the framework of the Four Variable Model. D. Parnas provided helpful insights about the Four Variable Model. N. Shankar suggested the algorithm that our tool uses to check for tautologies. R. Bharadwaj, M. Chechik, S. Faulk, J. Kirby, and the anonymous referees provided useful comments that led to significant improvements in our presentation. Finally, we gratefully acknowledge K. Britton, D. Parnas, and J. Shore, who developed the original A-7 requirements document, for their pioneering work in the area of requirements specification.

References

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

[2] Atlee, J. M., and Gannon, J. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng. 19*, 1 (Jan. 1993), 24–40.

[3] Berry, G., and Gonthier, G. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming 19* (1992), 89.

[4] Bharadwaj, R. A generalized validity checker. Tech. rep., Naval Research Lab., Wash., DC, 1996. In preparation.

[5] Boehm, B. W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers C-35*, 8 (Aug. 1986), 677–691.

[7] Clarke, E. M., Emerson, E., and Sistla, A. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang. and Systems 8*, 2 (Apr. 1986), 244–263.

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

[9] Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M. A tutorial introduction to PVS. Tech. rep., Computer Science Lab, SRI Int'l, Menlo Park, CA, Apr. 1995. (Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, FL).

[10] Dai, H., and Scott, C. K. AVAT, a CASE tool for software verification and validation. In *Proc. IEEE 7th Intl. Workshop on Computer-Aided Software Engineering–CASE'95* (Toronto, ON, Canada, July 1995), pp. 358–367.

[11] Fairley, R. *Software Engineering Concepts.* McGraw-Hill, New York, NY, 1985.

[12] Faulk, S. R. *State Determination in Hard-Embedded Systems.* PhD thesis, Univ. of NC, Chapel Hill, NC, 1989.

[13] Faulk, S. R. Software requirements: A tutorial. Tech. Rep. NRL-7775, Naval Research Lab., Wash., DC, 1995.

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

[15] Faulk, S. R., Finneran, L., Kirby, Jr., J., Shah, S., and Sutton, J. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94)* (Gaithersburg, MD, June 1994), pp. 3–8.

[16] Garey, M. R., and Johnson, D. S. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* Freeman, New York, NY, 1979.

[17] Grindley, C. B. B. The use of decision tables within Systematics. *The Computer J. 11*, 2 (Aug. 1968), 128–133.

[18] Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming 8*, 3 (June 1987), 231–274.

[19] Heimdahl, M. P. E., and Leveson, N. Completeness and consistency analysis of state-based requirements. In *Proc. of 17th Int'l Conf. on Softw. Eng. (ICSE '95)* (Seattle, WA, Apr. 1995), ACM, pp. 3–14.

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

[21] Heitmeyer, C., Labaw, B., and Kiskis, D. Consistency checking of SCR-style requirements specifications. In *Proc., International Symposium on Requirements Engineering* (Mar. 1995).

[22] Heitmeyer, C., and Mandrioli, D., Eds. *Formal Methods for Real-Time Computing.* Trends in Software. John Wiley & Sons Ltd, Chichester, England, 1996.

[23] Heitmeyer, C. L. Requirements specification for hybrid systems. In *Proceedings, Hybrid Systems Workshop III, Lecture Notes in Computer Science*, R. Alur, T. Henzinger, and E. Sontag, Eds. Springer-Verlag, Norwell, MA, 1996. To appear.

[24] Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G. Tools for analyzing SCR-style requirements specifications: A formal foundation. Tech. rep., Naval Research Lab., Wash., DC, 1996. In preparation.

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

[26] Heninger, K., Parnas, D. L., Shore, J. E., and Kallander, J. W. Software requirements for the A-7E aircraft. Tech. Rep. 3876, Naval Research Lab., Wash., DC, 1978.

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

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

[29] Hoover, D. N., and Chen, Z. Tablewise, a decision table tool. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)* (Gaithersburg, MD, June 1995), IEEE, pp. 97–108.

[30] Hughes, M. L., Shank, R. M., and Stein, E. S. *Decision Tables.* MDI Publications, Wayne, PA, 1968.

[31] Hurley, R. B. *Decision Tables in Software Engineering.* Van Nostrand Reinhold, New York, NY, 1983.

[32] Jaffe, M. S., Leveson, N. G., Heimdahl, M. P. E., and Melhart, B. E. Software requirements analysis for real-time process control systems. *IEEE Trans. Softw. Eng. SE-17*, 3 (Mar. 1991), 241–258.

[33] Janicki, R. Towards a formal semantics of Parnas tables. In *Proc. 17th Int'l Conf. on Softw. Eng. (ICSE '95)* (Seattle, WA, Apr. 1995), ACM, pp. 231–240.

[34] Landwehr, C. E., Heitmeyer, C. L., and McLean, J. A security model for military message systems. *ACM Trans. on Comp. Syst. 2*, 3 (Aug. 1984), 198–222.

[35] Leveson, N. G., Heimdahl, M. P., Hildreth, H., and Reese, J. D. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng. 20*, 9 (Sept. 1994).

[36] Lincoln, P., and Rushby, J. A formally verified algorithm for interactive consistency under a hybrid fault model. Tech. Rep. SRI-CSL-93-02, Computer Science Lab, SRI Int'l, Menlo Park, CA, Mar. 1993.

[37] Lutz, R. R. Targeting safety-related errors during software requirements analysis. In *Proc. 1st ACM SIGSOFT Symp. on Foundations of Softw. Eng.* (Los Angeles, CA, Dec. 1993).

[38] Metzner, J. R., and Barnes, B. H. *Decision Table Languages and Systems*. Academic Press, New York, NY, 1977.

[39] Meyer, S., and White, S. Software requirements methodology and tool study for A6-E technology transfer. Tech. rep., Grumman Aerospace Corp., Bethpage, NY, July 1983.

[40] Parnas, D. L. Tabular representation of relations. Tech. Rep. CRL–260, Telecommunications Research Institute of Ontario (TRIO), McMaster Univ., Hamilton, ON, Canada, Oct. 1992.

[41] Parnas, D. L. Some theorems we should prove. In *Proc., 1993 Int'l Conf. on HOL Theorem Proving and Its Applications* (Vancouver, BC, Canada, Aug. 1993), pp. 155–162.

[42] Parnas, D. L. Inspection of safety-critical software using program-function tables. Tech. Rep. CRL-288, Communications Research Laboratory, McMaster Univ., Hamilton, ON, Canada, 1994.

[43] Parnas, D. L., Asmis, G., and Madey, J. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety 32*, 2 (April–June 1991), 189–198.

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

[45] Parnas, D. L., and Madey, J. Functional documentation for computer systems. *Science of Computer Programming 25*, 1 (Oct. 1995), 41–61.

[46] Porter, A. A., and Votta, Jr., L. G. An experiment to assess different defect detection methods for software requirements inspections. In *Proc. 16th Int'l Conf. on Softw. Eng.* (1994).

[47] Rushby, J., and Srivas, M. Using PVS to prove some theorems of David Parnas. In *Proc., 1993 Int'l Conf. on HOL Theorem Proving and Its Applications* (Vancouver, BC, Canada, Aug. 1993), pp. 163–173.

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

[49] U. S. General Accounting Office. Mission critical systems: Defense attempting to address major software challenges. Tech. Rep. GAO/IMTEC–93–13, U. S. General Accounting Office, Wash., DC, Dec. 1992.

[50] van Schouwen, A. J. The A-7 requirements model: Re-examination for real-time systems and an application for monitoring systems. Tech. Rep. TR 90-276, Queen's Univ., Kingston, ON, Canada, 1990.

[51] van Schouwen, A. J., Parnas, D. L., and Madey, J. Documentation of requirements for computer systems. In *Proc. RE'93 Requirements Symp.* (San Diego, CA, Jan. 1993), pp. 198–207.