Consistency Checking of SCR-Style Requirements Specifications

Constance Heitmeyer and Bruce Labaw*

Daniel Kiskis†

Abstract

This paper describes a class of formal analysis called consistency checking that mechanically checks requirements specifications, expressed in the SCR tabular notation, for application-independent properties. Properties include domain coverage, type correctness, and determinism. As background, the SCR notation for specifying requirements is reviewed. A formal requirements model describing the meaning of the SCR notation is summarized, and consistency checks derived from the formal model are described. The results of experiments to evaluate the utility of automated consistency checking are presented. Where consistency checking of requirements fits in the software development process is discussed.

1 Introduction

A recent study of industrial application of formal methods concludes that formal methods, including those for specifying and analyzing requirements, are "beginning to be used seriously and successfully by industry... to develop systems of significant scale and importance" [5]. Included in the study is the Software Cost Reduction (SCR) method for specifying requirements. Introduced more than a decade ago to describe the functional requirements of software unambiguously and concisely [13, 14], the SCR method has been extended recently to describe system, rather than simply *software*, requirements and to incorporate techniques for representing nonfunctional requirements, such as timing and precision [17, 21, 22].

Designed originally for use by engineers, the SCR method has been successfully applied to a variety of practical systems. These include avionic systems, such as the A-7 Operational Flight Program (OFP) [13, 1]; a submarine communications system [12]; and safety-critical components of two nuclear power plants, the Darlington plant in Canada [22] and a second plant in Belgium [4]. Recently, a consortium of aerospace companies has developed a version of the SCR method, called CoRE, to capture and document the requirements of avionics and space applications [7, 8].

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. As observed in the formal methods study [5], tool support for formal methods, though currently weak and impoverished, is "necessary for the full industrialization process... and needs to be an integral part of a broader software development tool suite." Further, one of the original developers of the SCR method and a leader in the certification of the Darlington software cites the "need for tool support to make the process practical" [18].

An important question is what form tool support should take. To answer this question, our group is developing a prototype toolset for constructing and analyzing SCR-style requirements specifications. The toolset includes a *specification editor* for creating and editing formal requirements specifications, a *simulator* for symbolically executing the specifications, and *formal analysis* tools for testing the specifications for selected properties.

Three classes of formal analysis can be applied to requirements specifications. One class called *consistency checking*, the subject of this paper, tests that requirements specifications satisfy a formal requirements model. The requirements model describes the set of properties that all requirements specifications must satisfy. Hence, the properties tested by the consistency checker are independent of a particular application.

The other two classes of formal analysis require the successful completion of the first: both depend on a consistent (and complete) requirements specification. The second class checks the requirements specification for application properties. These properties include *safety properties*, which prevent unplanned events that result in death, injury, illness, or damage to property; *timing properties*, which require the system to produce results within specified time intervals (see, e.g., [9]); and *security properties*, which prevent the unauthorized disclosure, modification, and withholding of sensitive information (see, e.g., [15]). Given a system requirements specification and another system description (such as a software design or source code), the third class of formal analysis checks that the system description satisfies the requirements specification.

The properties that consistency checking tests are usually quite simple. For example, given a requirements specification that includes a total function F, the consistency checker tests that F is indeed total (i.e., defined everywhere in F's domain). While simple, the number of times such 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,

*Code 5546, Naval Research Lab, Wash., DC 20375.

†3060 Whisperwood Drive, Ann Arbor, MI 48105.

0-8186-7017-7/95 $04.00 © 1995 IEEE

safety-relevant issues" [18]. 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 have a formal (that is, mathematical) foundation and 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 has a formal foundation, namely, our formal requirements model [10]. It is easy to use: after developing a requirements specification in the SCR notation, the engineer 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 in the requirements specification of a medium-size Navy application. These errors were detected even though the specification had previously undergone comprehensive checks by two independent review teams. These results and the high cost of the Darlington certification effort, where such checks were done by hand, suggest that automated consistency checking is cost-effective. Finally, automated consistency checking is an important first step in formal analysis of requirements specifications, since, as indicated above, other classes of formal analysis require a consistent specification.

Although earlier requirements models, in particular, Faulk's automaton model [6], Parnas' Four-Variable Model [17], and the model underlying van Schouwen's specification [21, 22], define some aspects of the SCR notation, these models are too abstract to provide a formal basis for our tools. To provide a precise and detailed semantics for the SCR notation, our requirements model represents the system to be built as a state automaton and describes the monitored and controlled variables, conditions, events, and other constructs that make up an SCR specification in terms of that automaton. This automaton model, an extension of Faulk's and a special case of the other two models, provides the formal basis for our automated consistency checker as well as our other tools, in particular, the specification editor, the simulator, and a *verifier* that checks the specifications for application properties (the second class of formal analysis described above).

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 where consistency checking fits in the software development process. The contributions of this paper are its introduction and formal definition of a class of analysis, namely, consistency checking, for detecting application-independent errors in system and software requirements specifications and the evidence it provides that software tools for automated consistency checking are useful and cost-effective.

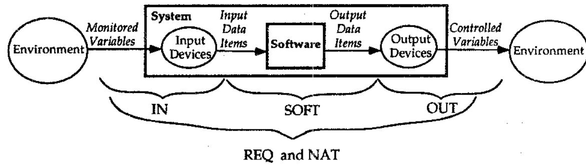

Figure 1: Four Variable Model.

2 Background: SCR Method

The purpose of a requirements document is to describe all acceptable system implementations [12]. To minimize implementation bias, a requirements document should specify only the externally visible behavior required of the system. To achieve this, Parnas has introduced the Four Variable Model, a standardarized model of embedded system behavior that describes the required system functions, timing, and precision [17]. This section reviews the constructs and tabular notation 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. The Four Variable Model, illustrated in Figure 1, represents requirements as a set of mathematical relations on four sets of variables called monitored, controlled, input, and output. A *monitored variable* represents an environmental quantity that influences system behavior, a *controlled variable* an environmental quantity the system controls. A black box specification of required behavior is given as two relations (REQ and NAT) from the monitored quantities to the controlled quantities (not inputs to outputs). NAT defines the set of possible values; it captures any constraints on behavior, such as those imposed by physical laws. REQ defines the additional constraints imposed by the system to be built. It describes the required system behavior by defining the relation the system must maintain between the monitored and the controlled quantities.

Inputs and outputs are treated as resources. Inputs are resources available to the system to compute the monitored quantities. The relation IN defines the mapping from the monitored quantities to the inputs. Similarly, the relation OUT defines the mapping from the outputs to the controlled quantities. The use of monitored and controlled quantities to define required behavior, rather than inputs and outputs, keeps the specification in the problem domain and allows a simpler specification. Below, we refer to monitored variables and inputs as *input variables*, controlled variables and outputs as *output variables*.

Four more constructs, all introduced in the A-7 requirements document [13], are useful for specifying systems using the Four Variable Model. These are modes, terms, conditions, and events. A *mode class* is a state machine, whose states are called *system modes* (or simply *modes*) and whose transitions are triggered by events. Complex systems are defined by more than one mode class, operating in parallel. A *term* is any function of input variables, modes, or other terms. A

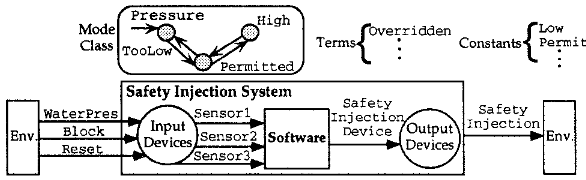

Figure 2: Requirements Spec. for Safety Injection.

condition is a predicate about the system state. An *event* occurs when any system *entity* (that is, an input or output variable, mode, or term) changes value. A special event, called an *input event*, occurs when an input variable changes value. Another special event, called 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 [4]. 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 could be used to specify the requirements of the control system. 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; and the hardware interface between the (control system) software and the safety injection system as an output.[1]

A mode class **Pressure** and a term **Overridden** help make the specification concise. **Pressure** contains three modes, **TooLow**, **Permitted**, and **High**. 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** is *true* if safety injection is blocked, *false* otherwise. An example of a condition in the specification is "**WaterPres** < **Low**". 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**).

SCR Notation. The A-7 requirements document [13] introduced a special tabular notation for writing specifications. Because tables are easy to understand, the tabular notation facilitates industrial application of the SCR method. Among the tables in SCR specifications are condition tables, event tables, and mode transition tables. Each table defines a function.[2] A condition table describes an output variable or a term as a function of a mode and a condition, an event table describes either 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.

[1] The example omits the SCR brackets, e.g., *mode*, etc.

[2] Although SCR specifications can be nondeterministic, our initial model is restricted to deterministic systems.

Old Mode	Event	New Mode
TooLow	@T(WaterPres ≥ Low)	Permitted
Permitted	@T(WaterPres ≥ 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 or Permitted	@T(Block=On) WHEN Reset=Off	@T(Inmode) OR @T(Reset=On)
Overridden	True	False

Table 2: Event Table for **Overridden**.

While condition tables define total functions, event tables and mode transition tables may define partial functions, because some events cannot occur when certain conditions are true. For example, in the above system, 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.

Tables 1–3 are part of REQ, the system requirements specification for the above 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**. Table 2 is an event table describing the term **Overridden** as a function of **Pressure**, **Block**, and **Reset**. 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 notation "@T(Inmode)" in a row of an event table describes system entry into the mode in that row; for example, "@T(Inmode)" in the first row of Table 2 means, "If the system enters **High**, then **Overridden** becomes *false*."

3 Formal Requirements Model

Our requirements model, a state automaton model, defines sets of modes, entity names, values, and types. It also introduces a function TY, which maps an entity to its legal values; in the sample system, TY(**Overridden**)={*true*, *false*}, TY(**Sensor1**)=[14, 2000], TY(**Safety Injection**)={**On**, **Off**}, and TY(**Pressure**)={**High**, **TooLow**, **Permitted**}. 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 that triggers a new system state. It then shows how a set of functions, called table functions, can be derived from the SCR tables. The table functions are used to define the system transform T, a special case of REQ which maps the current system state and an input event to a new system state. To provide a formal foundation for

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

Table 3: Condition Table for `Safety Injection`.

consistency checking, we present below excerpts from our requirements model [10].

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

- *MS* is the union of N pairwise disjoint sets, 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* is a set of entity values with $VS = MS \cup TS$.
- *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 of entity r.

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.

Conditions. Conditions are defined on the values of entities in *RF*. A *simple condition* c is either *true*, *false*, or a logical statement $c = r \odot v$, 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. A *condition* c is a logical statement composed of simple conditions connected in the standard way by the logical connectives $\wedge$, $\vee$, and NOT.

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

- E^m is a set of input events. A *primitive event* is denoted as @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.
- S is the set of possible system states.
- s_0 is a special state called the initial state.
- T is the system transform, i.e., a function from $E^m \times S$ into S.

Events. In addition to denoting primitive events, the "@T" notation also denotes basic events and conditioned events. A *basic event* e is denoted as $e =$ @T(c), where c is any simple condition. A *simple conditioned event* e is denoted as $e =$ @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 $e =$ @T(c) can be expressed as the simple conditioned event $e =$ @T(c) WHEN *true*. A *conditioned event* e is composed of simple conditioned events connected by the logical connectors $\wedge$ and $\vee$.

We define the logical statement represented by a simple conditioned event as @T(c) WHEN d = NOT $c \wedge c' \wedge d$, where the unprimed and primed versions of condition c denote c in different states. We define c', where $c = r \odot v$, 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. Where a condition is evaluated in a single state, an event is evaluated in two states: unprimed conditions in the first (or old) state, primed conditions in the second (or new) state.

Ordering the Entities. To compute the value of an entity 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, we associate with each entity r a subset of *RF* called the *new state dependencies* set. Given entities r and $\hat{r}$ in *RF*, we say that r *depends directly on* $\hat{r}$ if $\hat{r}$ is in r's new state dependencies set. The "depends directly on" relation imposes a partial ordering on the set *RF*. Thus, 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, $i > j$).

Table Functions. Each SCR table describes a *table function*, called F_i, for some entity r_i. Table functions define the values of the output variables, terms, and mode classes in SCR requirements specifications. Each entity defined by a table is associated with exactly one mode class, M_j, $1 \leq j \leq N$. To represent the relation between an entity and a mode class, we define a function μ, where $\mu(i) = j$ iff 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.

Presented below for condition, event, and mode transition tables is a typical format, a representation of the information in each table as a relation ρ_i, and a set of properties which guarantee that ρ_i is a function. Given ρ_i, we can derive the table function F_i.

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 between modes, conditions, and values, i.e., $\rho_i = \{(m_j, c_{j,k}, v_k) \in M_{\mu(i)} \times C_i \times TY(r_i)\}$, where C_i is a set of conditions defined on entities in RF. ρ_i has 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 are included).
3. For all j: $\vee_{k=1}^{p} c_{j,k} = true$ (Coverage: The disjunction of the conditions in each row of the table is *true*).
4. For all j, k, l, $k \neq l$: $c_{j,k} \wedge c_{j,l} = false$ (Disjointness: The conjunction of the conditions in each row of the table is *false*).

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

Table 4: Condition Table—Typical Format.

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

Table 5: Event Table—Typical Format.

These properties guarantee that ρ_i is a function.

To make explicit entity r_i's dependencies on other entities, we consider an alternate form F_i of the function ρ_i. To define F_i, we require the new state dependencies set, $D_i^n = \{y_{i,1}, y_{i,2}, \ldots, y_{i,n_i}\}$, where $y_{i,1}$ is the entity name for the associated mode class. Based on D_i^n and ρ_i, we define F_i as

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

The function F_i is called a *condition table* function. The four properties guarantee that F_i is total.

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. ρ_i has the following two properties:

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

These properties and assumptions on input events guarantee that ρ_i is a function.

As with condition tables, we make explicit r_i's dependency on other entities by defining an alternate form F_i of the function ρ_i. To define F_i, we require both the new state dependencies set D_i^n and an *old state dependencies* set $D_i^o = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}\}$, where $D_i^o \subseteq RF$ contains the entities needed in the old state to compute r_i and $x_{i,1}$ is the entity name for the associated mode class. Based on D_i^o, D_i^n, and ρ_i, F_i is defined by

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

Table 6: Mode Transition Table—Typical Format.

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

The function F_i is called an ***event table*** function.

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

1. The m_j are unique.
2. For all $k \neq k'$, $m_{j,k} \neq m_{j,k'}$, and for all j and for all k, $m_j \neq m_{j,k}$.
3. For all j, k, k', $k \neq k'$: $e_{j,k} \wedge e_{j,k'} = false$ (Determinism: The conjunction of the conditioned events in each row of the table is *false*).
4. For all $m \in M_{\mu(i)}$, there exists j such that $m_j = m$ or there exist j and k such that $m_{j,k} = m$ (Each mode in the mode class is in either ρ_i's domain or its image).

These properties and assumptions on input events guarantee that ρ_i is a function. It is easy to show that a mode transition table with the format shown in Table 6 can be expressed in the format shown for an event table. Hence, a mode transition table can be expressed as an event table function F_i.

Example. To illustrate the formal model, we consider the condition table shown in Table 3 for the controlled variable `Safety Injection`. The new state dependencies set for `Safety Injection` is D_i^n = {`Pressure`, `Overridden`}, where i is the index of `Safety Injection` in the sequence R. Because it depends directly on `Pressure` and `Overridden`, `Safety Injection` follows them in R. The condition table function F_i for `Safety Injection` is defined by

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

4 Automated Consistency Checking

Listed below are examples of consistency checks derived from our requirements model.

- **Proper Syntax.** Each component of the specification has proper syntax. For example, each condition and event is well-defined.
- **Type Correctness.** All type definitions are satisfied, each entity is assigned a type, and all types are defined.
- **Completeness.** The value of each output variable, term, and mode class is defined. (Most variables will be defined by tables, but standard mathematical definitions may be given for some output variables and terms.)
- **Reachability.** No mode is unreachable.
- **Initial values.** Initial values are defined for all mode classes and input variables *and* for all terms and output variables not defined by condition tables. (Initial values are not required for entities defined by condition tables, since they can be derived from the tables.)
- **Consistency.** Each condition table, event table, and mode transition table satisfies the appropriate properties in Section 3.
- **Lack of Circularity.** There are no circular dependencies.

Clearly, some checks must precede others. For example, checks for proper syntax must precede type checking, and type checking should precede checking that a total function is indeed total.

Examples. Checking the consistency of Table 7, a modification of the condition table in Table 3, reveals four errors. The second row violates both Coverage (Overridden $\vee$ Overridden $\neq$ *true*) and Disjointness (Overridden $\wedge$ Overridden $\neq$ *false*). The third row has two type errors: Safety Injection has the values Off and On, not False and True.

Determinism, the second property required of event tables, is violated if events in two different columns, say e and e', overlap, i.e., $e \wedge e' \neq$ *false*. To check the second row of Table 8 (a variation of the event table in Table 2) for Determinism, the expression, [@T(Block=On) WHEN Reset=Off] $\wedge$ [@T(Block=On) $\vee$ @T(Reset=On)], is evaluated. This expression can be rewritten as a disjunction, [@T(Block=On) WHEN Reset=Off $\wedge$ @T(Block=On)] $\vee$ [@T(Block=On) WHEN Reset=Off $\wedge$ @T(Reset=On)]. Applying the definition of event evaluation in Section 3 to the first clause of the disjunction, we have [Block$'$ =On $\wedge$ Block=Off $\wedge$ Reset=Off] $\wedge$ [Block$'$ =On $\wedge$ Block=Off]. This implies Block$'$ =On $\wedge$ Block=Off $\wedge$ Reset=Off. Because this expression does not equal *false*, the specified behavior is nondeterministic. Thus, 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*.

Mode	Conditions	
High or Permitted	True	False
TooLow	Overridden	Overridden
Safety Injection	False	True

Table 7: Modified Table for Safety Injection.

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

Table 8: Modified Table for Overridden.

Some checks, such as syntax and type checking, are easy. More complex are checks that evaluate expressions containing "Inmode", depend on nonlocal definitions (other than type information), or 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, i.e., if @T(WaterPres$\geq$Permit) $\wedge$ @T(WaterPres<Low) is *true*. This implies WaterPres$'$ $\geq$ Permit $\wedge$ WaterPres $\not\geq$ Permit $\wedge$ WaterPres$'$ < Low $\wedge$ WaterPres $\not<$ Low. By assumptions on the constants, Permit > Low. This and WaterPres$'$ $\geq$ Permit imply WaterPres$'$ $\geq$ Low. Hence, the expression is *false* and the defined behavior deterministic. Because in general mechanical evaluation of such expressions is hard, the tool may need some feedback from the user to complete certain checks.

Prototype Consistency Checker. A prototype consistency checker that performs most of the above checks has been implemented. It 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 errors that it finds. The user may select one of the listed errors. In response, the tool displays the part of the specification that contains the error, so that the user can make needed corrections.

5 Applying Consistency Checks

To evaluate the utility of checking requirements specifications for consistency, we conducted two experiments. In the experiments, we used early versions of the consistency checker to analyze tables in a revision [1] of the software requirements document for the A-7's Operational Flight Program (OFP). The new document corrects errors in the original [13] and uses Faulk's tabular format to specify mode transitions [6].

In the first experiment, our tool tested all 36 condition tables in [1], a total of 98 rows, for Coverage and Disjointness. The tool found 19 errors. Seventeen of these, distributed over 11 tables, proved to be legiti-

Error	No.	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` ∨ `Range to RMax` < 0) and NOT (`range to target` ≤ 10 mi.) do not cover the domain.

Table 9: Errors in the A-7 condition tables.

mate errors. (Classifying the remaining two as correct required information about the specification that our simple tool lacked.) Table 9 describes the detected errors. Interestingly, all are Coverage errors.

In a second experiment, our tool checked all mode transition tables in [1] for nondeterminism. The A-7 specification contains three mode classes with a total of 46 different modes (18 modes in the first mode class, 7 modes in the second, and 21 modes in the third). The tool checked 688 rows and found 33 nondeterministic transitions. Although many of these are undoubtedly errors, a few probably are not, since some detected events may be impossible. (Recall that some events cannot occur when certain conditions are true.) Reference [11] contains examples of nondeterminism our tool detected in the transition table for the OFP's Alignment, Navigation and Test mode class.

Tool-based vs. Manual Checks. Prior to publication, the revised A-7 requirements document was carefully reviewed by two teams, one made up of NRL computer scientists (including one of the authors), 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 those we developed, 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 listed in Section 4. Not only are tools more effective than people for checking these properties; in addition, they can reduce significantly 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 (the third class of analysis described in the introduction). 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 kg. [18]. Using tools to do such analyses should cost far less than using people.

Related Work. In a related effort, Atlee and Gannon use model checking to analyze SCR requirements specifications [2]. Unlike our tool, theirs evaluates application-specific properties. Further, where our consistency checker tests all tables and definitions in an SCR specification automatically, their tool analyzes the mode transition tables only, extended by hand to incorporate the needed variable definitions.

In other related work, Parnas describes ten small theorems related to his tabular notation (similar to other SCR notation) and challenges the developers of automated proof systems to prove the theorems [18]. 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 [20], 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 [16]. That PVS can prove such theorems easily is not too surprising, since the proofs require very simple logic. What is noteworthy about the PVS experiment is that the theorems were proven automatically.

A recent experiment [19] 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 would probably detect more errors than either alone.

6 Software Development Process

We envision the following process for developing requirements specifications. First, the developer uses the SCR notation to specify the requirements. Next, he uses an automated consistency checker to test for syntax and type correctness, coverage, determinism, and other application-independent properties. The next step is to symbolically execute the specification, using a simulator, to ensure that it captures the developer's intent; the simulator can be run either manually, or automatically using an input script (see, e.g., [3]).

In the later stages of requirements, the developer uses mechanical support to analyze the specification for application properties. Initially, he extracts a small subset with fixed parameters and only a few states from the specification and uses a model checker. This may be repeated, each time with a different or larger subset. Once he has sufficient confidence in the specification, the developer may use a deductive proof system to verify safety-critical components.

7 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 unpleasant task of checking specifications for consistency.
- Properly designed tools are significantly more cost-effective than people for consistency checking.
- 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 methods on which our tools are based scale up. They detected a significant number of errors in a medium-size real-world specification.

Currently, we are building a more complete version of the toolset, which includes the consistency checker, a specification editor and a simulator. We also plan a verifier that checks 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 the logic ourselves.

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 significantly reduce software development costs.

Acknowledgments

We gratefully acknowledge the work of A. Bull, C. Gasarch, and A. Rose on the consistency checker. R. Jeffords helped define the formal model. D. Berry gave valuable suggestions on an earlier draft. S. Faulk provided Fig. 1 and valuable comments that significantly improved the paper's presentation and its content.

References

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

[2] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. In *Proc., ACM SIGSOFT Conf. on Software for Critical Systems*, New Orleans, December 1991.

[3] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose. MT: A toolset for specifying and analyzing real-time systems. In *Proc., Real-Time Systems Symp.*, Raleigh, NC, December 1993.

[4] P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc., 15th Intern. Conf. on Software Eng.*, Baltimore, 1993.

[5] D. Craigen et al. An international survey of industrial applications of formal methods. Technical Report NRL-9581, NRL, Wash., DC, 1993.

[6] S. Faulk. *State Determination in Hard-Embedded Systems.* PhD thesis, Univ. of No. Carolina, Chapel Hill,, 1989.

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

[8] S. R. Faulk, L. Finneran, J. Kirby, and J. Sutton. Experience applying the CoRE method to the Lockheed C-130J. In *Proc., Ninth Annual Conf. on Computer Assurance*, Gaithersburg, MD, June 1994.

[9] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, December 1994.

[10] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical Report NRL-7499, NRL, Wash., DC, 1995. In preparation.

[11] C. L. Heitmeyer and B. G. Labaw. Consistency checks for SCR-style requirements specifications. Technical Report 9586, NRL, Wash DC, December 1993.

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

[13] K. Heninger, D. Parnas, J. Shore, and J. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, NRL, Wash., DC, 1978.

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

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

[16] S. Owre, N. Shankar, and J. Rushby. User guide for the PVS specification and verification system (Draft). Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.

[17] D. Parnas and J. Madey. Functional documentation for computer systems engineering (Version 2). Technical Report CRL 237, Telecommunications Research Inst. of Ontario (TRIO), McMaster Univ., Hamilton, Ont., 1991.

[18] D. L. Parnas. Some theorems we should prove. In *Proc., 1993 Intern. Conf. on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.

[19] A. A. Porter and L. G. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proc., 16th Intern. Conf. on Software Eng.*, 1994.

[20] J. Rushby and M. Srivas. Using PVS to prove some theorems of David Parnas. In *Proc., 1993 Intern. Conf. on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.

[21] A. J. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application for monitoring systems. Technical Report TR 90-276, Queen's Univ., Kingston, Ont., 1990.

[22] A. J. van Schouwen, D. L. Parnas, and J. Madey. Documentation of requirements for computer systems. In *Proc., RE'93 Requirements Symp.*, San Diego, January 1993.