ACM Transactions on Software Engineering and Methodology, 1996.

Automated Consistency Checking of Requirements Specifications

CONSTANCE L. HEITMEYER,
RALPH D. JEFFORDS,
BRUCE G. LABAW

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

2010.09.09

Abstract

This article 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 of two experiments are presented 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.

Contents

1. Introduction

- Errors in requirements are pervasive, dangerous, and costly.
- 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 <u>formal methods</u> during the requirements phase of software development.
  - Formal specification
  - Formal analysis

1. Introduction

- 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.

- Designed for use by engineers, the SCR method has been successfully applied to a variety of practical systems
  - avionics systems, such as the A-7 Operational Flight Program
  - a submarine communications system
  - safety-critical components of the Darlington nuclear power plant in Canada

- While the above applications of SCR rely mostly on manual techniques, effective use of the method in industrial settings will require powerful and robust tool support.

1. Introduction

- 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.

- We are developing a suite of prototype tools for the SCR
  - Specification editor
  - Simulator
  - Formal analysis tools
    - Consistency checker : for domain-independent properties
    - Verifier : for critical application properties

1. Introduction

- Although the consistency checking 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.

- An industrial-strength formal method should be
  - Usable by engineers,
  - Scalable, and
  - Cost effective.

- Automated consistency checking as described in this article is an important step in developing such a method for requirements specification.

2. Review of the SCR Method
2.1 Background

- The purpose of a requirements document is to describe all acceptable system implementations.

- 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.

- Faulk [1989] provided formal definitions for parts of the A-7 model.

- Using the original A-7 requirements document as a model, van Schouwen [1990] published a system-level requirements specification for the Water Level Monitoring System (WLMS), part of the shutdown system for a nuclear power plant

- The Four-Variable Model of Parnas and Madey [1995] provides a formal framework for the SCR method.

2.2 Four-Variable Model

- The Four-Variable Model, 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.

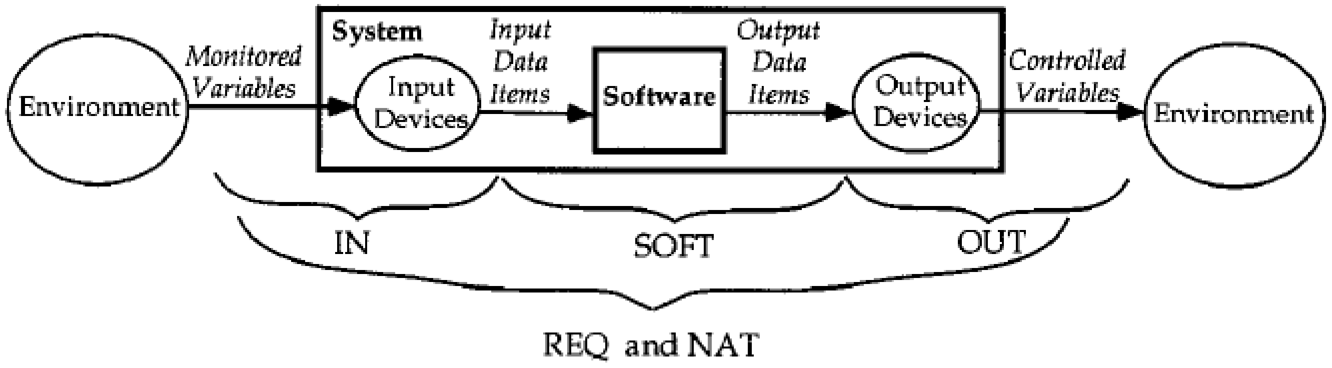


Fig. 1. The Four-Variable Model.

2.2 Four-Variable Model

- 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 y describing SOFT, the required relation between the input and output data items.

2.3 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 [Heninger et al. 1978], are useful.

- 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.

- A *term* is an auxiliary function defined on input variables, modes, or other terms 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.

2.3 SCR Constructs

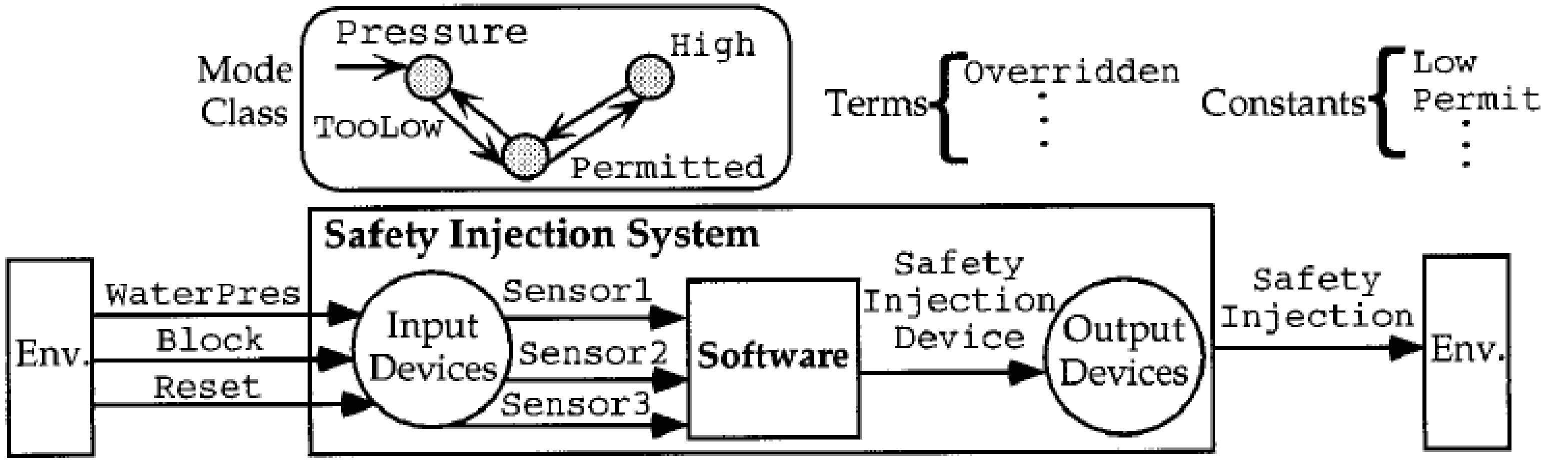


Fig. 2. Requirements specification for Safety Injection.

@T(Block=On)
→ (the operator turns Block from Off to On)

@T(Block=On) WHEN WaterPres < Low
→ (the operator turns Block to On when water pressure is below Low).

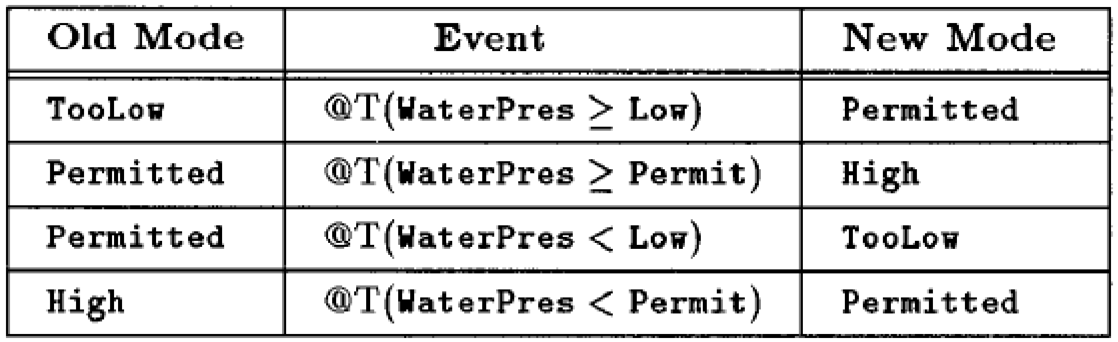
2.4 SCR Tables

- 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.

- Each table defines a mathematical function.
  - Condition table
  - Event table
  - Mode transition table

2.4 SCR Tables

Table I. Mode Transition Table for Pressure

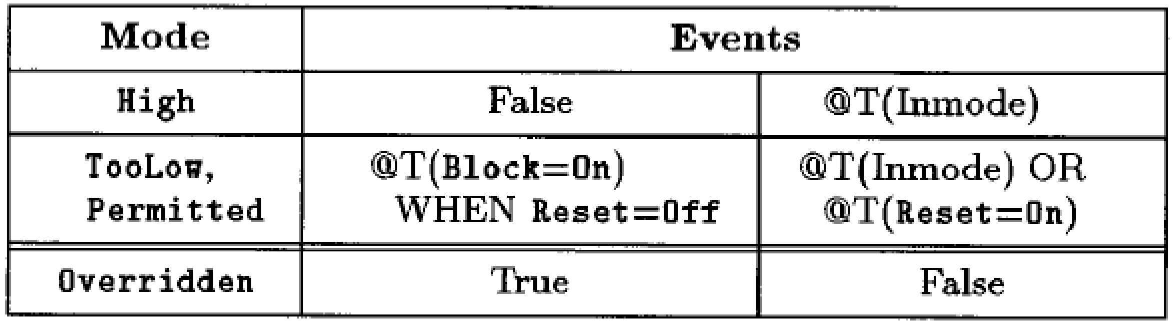


Old Mode	Event	New Mode
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

"If Pressure is TooLow, and WaterPres rises to Low, then Pressure changes to Permitted."

2.4 SCR Tables

Table II. Event Table for Overridden



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

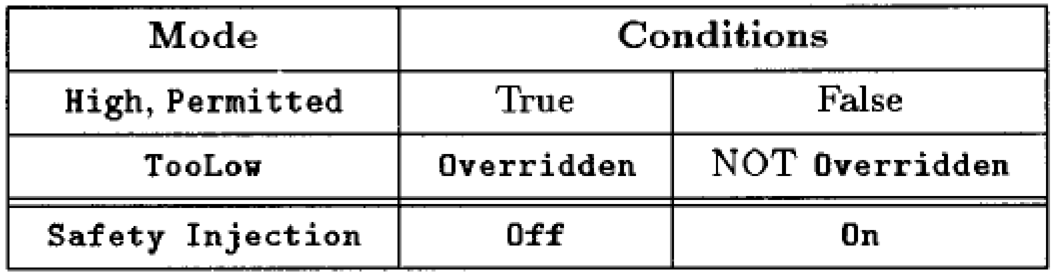
“If Pressure is TooLow, and Block becomes On when Reset is Off, then Overridden becomes *true*.”

The entry “False” in row 1 of Table II means that when the mode class is High no event can cause Overridden to become *true.*

“@T(Inmode)” in the second row of Table II means, “If the system enters TooLow or Permitted, then Overridden becomes false.”

2.4 SCR Tables

Table III. Condition Table for SafetyInjection



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

“If Pressure is High or Permitted, or if Pressure is TooLow and Overridden is true, then SafetyInjection is Off;
if Pressure is TooLow, and Overridden is false, then SafetyInjection is On.”

The entry “False” in the first row means that SafetyInjection is never On when Pressure is High or Permitted.

2.4 SCR Tables

- 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.
  - 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

- 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.

3.1 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, . . . , M_N, called *mode classes*. Each member of a mode class is called a *mode*.

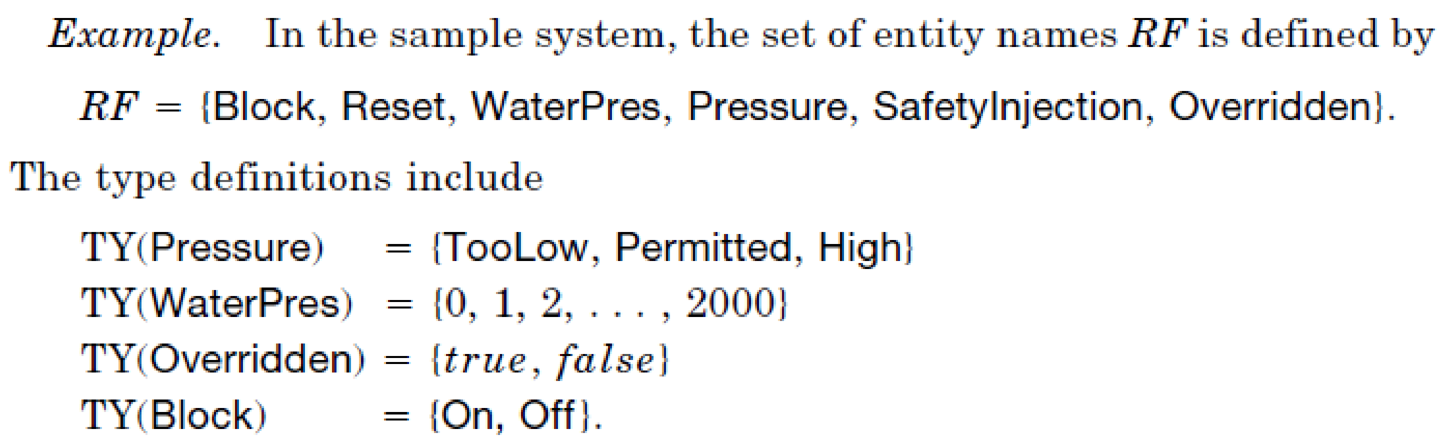
—TS is a union of data types, where each type is a nonempty set of values.

—$VS = MS \cup TS$ is the set of entity values.

—RF is a set of entity names. 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.

3.1 System State

- 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.



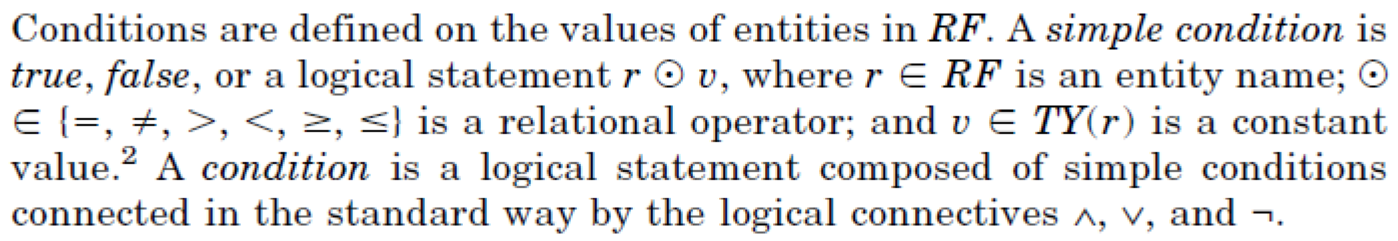
Example. In the sample system, the set of entity names RF is defined by

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

The type definitions include

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

3.2 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 $\neg$.

3.3 Events

- The "$@T$" notation denotes various events.

- Primitive event : $@T(r = v)$
- Input event: if $r \in IR$
- Basic event: $@T(c)$, where c is any simple condition
- Simple conditioned event: $@T(c)$ *when* $d \equiv @T(c)$ *when* $d = \neg c \wedge c' \wedge d$
- Conditioned event: composed of simple conditioned events connected by the logical connectors $\vee$ and $\wedge$.

- Any conditioned event can be expressed as a logical statement.

3.4 System (Software System)

A *system* (*software system*) is a 4-tuple, $= (E^m, S, s_0, T)$, where

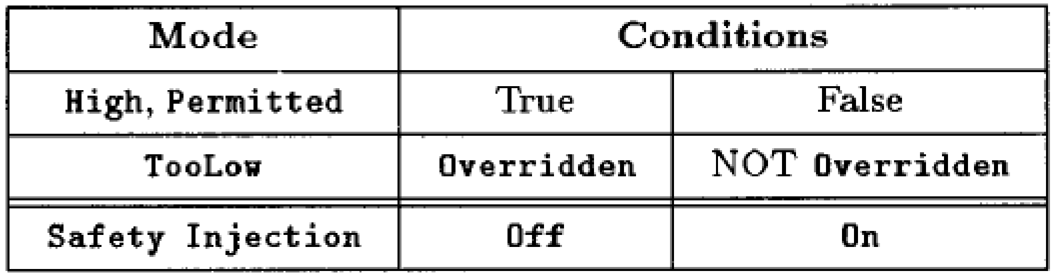
—E^m is a set of input events,
—S is the set of possible system states,
—s_0 is a special state called the initial state, and
—T, the system transform, is a partial function[3] from $E^m \times S$ into S.

- A basic assumption, called the <u>One-Input Assumption</u>, is that exactly one input event occurs at each state transition.

3.5 Ordering the Entities

- Partial ordering on all entities in the set *RF*.
- Example.

Table III. Condition Table for SafetyInjection



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

  - Partial ordering
    R = < WaterPres, Block, Reset, Pressure, Overridden, SafetyInjection>

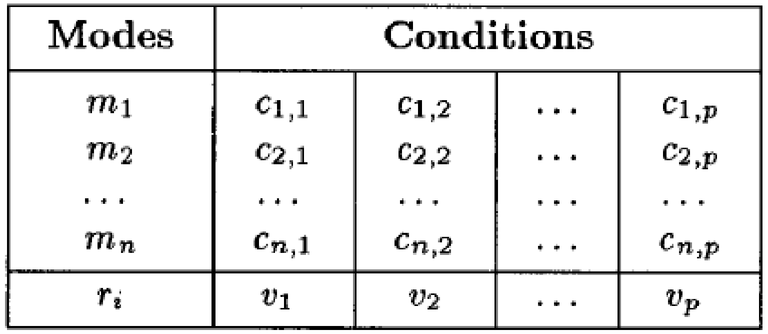
3.6 Table Functions

- Each SCR table describes a table function, called F_i, which defines an output variable, a term, or a mode class r_i.

3.7 Condition Tables

- Each condition table describes an output variable or term r_i as a relation ρ_i defined on modes, conditions, and values.

Table IV. Condition Table—Typical Format



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

- 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*).

3.7 Condition Tables

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

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

- Example. Based on the new state dependencies set {Pressure, Overridden} and Table III, the condition table function for SafetyInjection, denoted F_6, is defined by

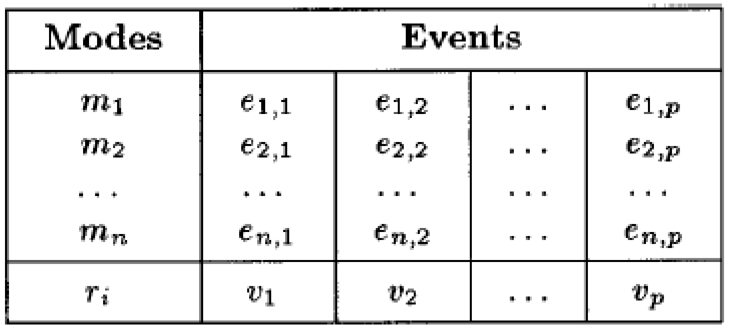
$$\text{SafetyInjection} =$$

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

3.8 Event Tables

- Each event table describes an output variable or term r_i *as* a relation ρ_i between modes, conditioned events, and values.

Table V. Event Table—Typical Format



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

- The relation ρ_i must satisfy the following four properties:

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

3.8 Event Tables

- 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.

- The function F_i is called an event table function.

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

3.8 Event Tables

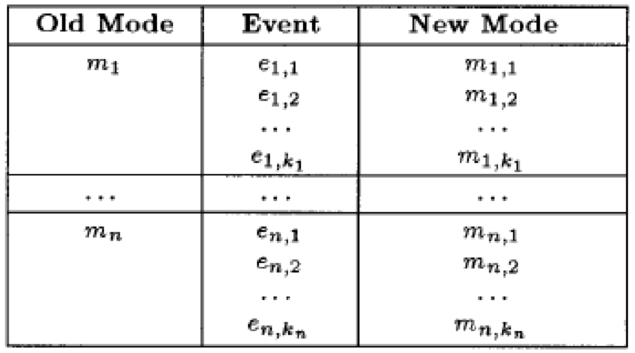
- Example. Both the old state dependencies set and the new state dependencies set for Overridden, {Block, Reset, Pressure, Overridden} and {Block, Reset, Pressure}, can be derived from Table II. Based on these sets and Table II, the event table function for Overridden is defined by

$$
\begin{aligned}
&\text{Overridden}' = \\
&F_5(\text{Block}, \text{Reset}, \text{Pressure}, \text{Overridden}, \text{Block}', \text{Reset}', \text{Pressure}') = \\
&\quad \begin{cases}
\mathit{true} & \text{if} \quad (\text{Pressure} = \text{TooLow} \wedge \text{Block}' = \text{On} \wedge \text{Block} = \text{Off} \wedge \\
& \qquad \text{Reset} = \text{Off}) \vee (\text{Pressure} = \text{Permitted} \wedge \text{Block}' = \text{On} \wedge \\
& \qquad \text{Block} = \text{Off} \wedge \text{Reset} = \text{Off}) \\
\mathit{false} & \text{if} \quad (\text{Pressure} = \text{TooLow} \wedge \text{Reset}' = \text{On} \wedge \text{Reset} = \text{Off}) \vee \\
& \qquad (\text{Pressure} = \text{Permitted} \wedge \text{Reset}' = \text{On} \wedge \text{Reset} = \text{Off}) \vee \\
& \qquad (\text{Pressure}' = \text{High} \wedge \text{Pressure} \neq \text{High}) \vee \\
& \qquad ((\text{Pressure}' = \text{Permitted} \vee \text{Pressure}' = \text{TooLow}) \wedge \\
& \qquad \neg (\text{Pressure} = \text{Permitted} \vee \text{Pressure} = \text{TooLow})) \\
\text{Overridden} & \text{otherwise}
\end{cases}
\end{aligned}
$$

3.9 Mode Transition Tables

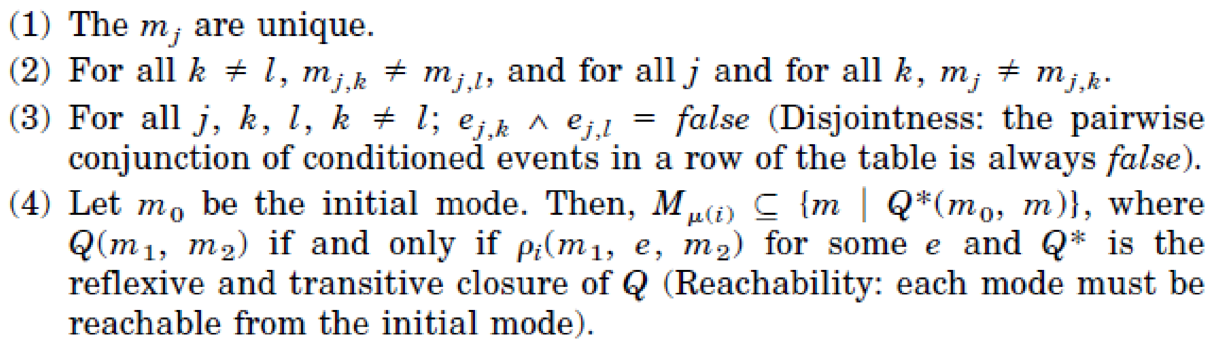
- Table VI shows a typical format for a mode transition table for an entity r_i that names a mode class $M_{\mu(i)}$.

Table VI. Mode Transition Table—Typical Format



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}

- The relation r_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).

3.9 Mode Transition Table

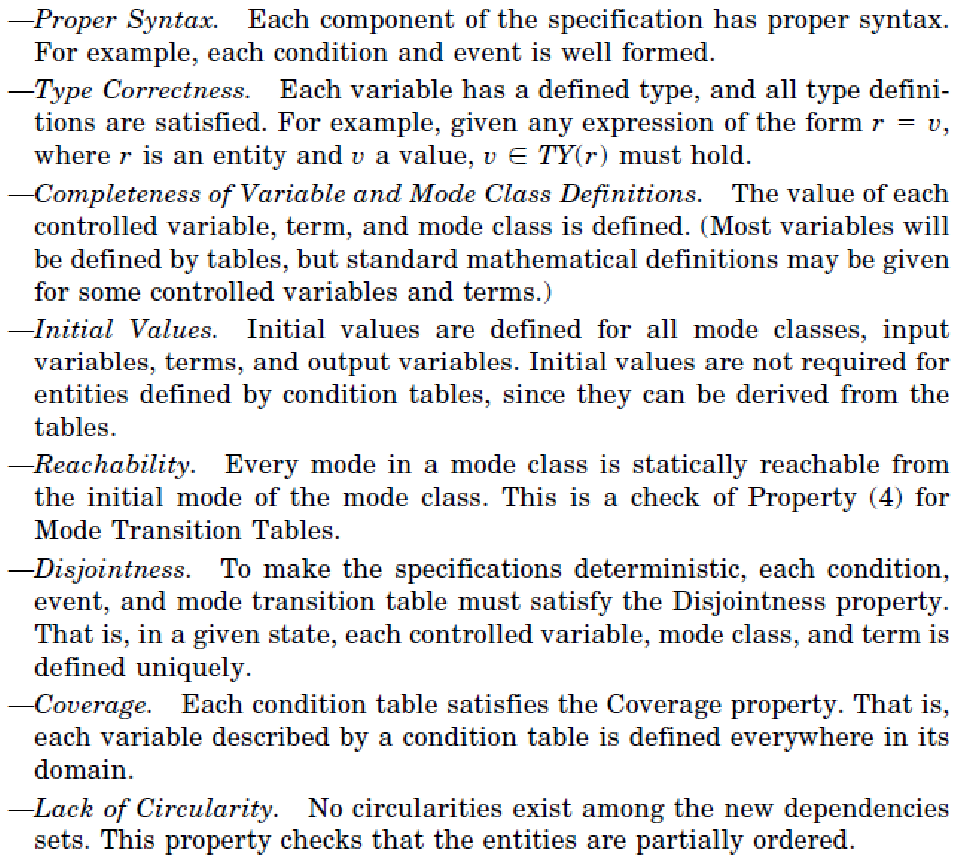
- It is easy to show that a mode transition table with the format in Table VI can be expressed in the format shown in Table V for an event table. Hence, a mode transition table can be expressed as an event table function.

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

$$
\begin{aligned}
&\text{Pressure}' = \\
&F_4\,(\text{Pressure}, \text{WaterPres}, \text{WaterPres}') = \\
&\begin{cases}
\text{TooLow} & \text{if} \quad \text{Pressure} = \text{Permitted} \wedge \text{WaterPres}' < \text{Low} \wedge \\ & \qquad \text{WaterPres} \nless \text{Low} \\
\text{High} & \text{if} \quad \text{Pressure} = \text{Permitted} \wedge \text{WaterPres}' \geq \text{Permit} \wedge \\ & \qquad \text{WaterPres} \ngeq \text{Permit} \\
\text{Permitted} & \text{if} \quad (\text{Pressure} = \text{TooLow} \wedge \text{WaterPres}' \geq \text{Low} \wedge \\ & \qquad \text{WaterPres} \ngeq \text{Low}) \vee (\text{Pressure} = \text{High} \wedge \\ & \qquad \text{WaterPres}' < \text{Permit} \wedge \text{WaterPres} \nless \text{Permit}) \\
\text{Pressure} & \text{otherwise.}
\end{cases}
\end{aligned}
$$

4. Automated Consistency Checking

- Consistency checks
  - determine whether the specifications are well formed
  - independent of a particular system state → 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 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.)

—*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 statically 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.

4.1 Checking for Disjointness and Coverage

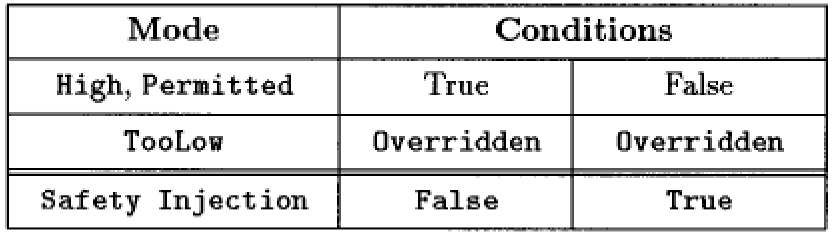
- The most computationally expensive checks are checks for Disjointness and Coverage.
- To check these properties, the consistency checker determines whether a logical expression is a tautology.
  - To determine whether these logical expressions are tautologies, our tool applies a tableaux-based decision procedure that encodes the algorithm in Smullyan [1968].

- 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 $c_1 \vee c_2 \vee \dots c_n = \mathit{true}$.

4.1 Checking for Disjointness and Coverage

- Example. Checking the consistency of Table VII, a modification of the condition table in Table III, reveals four errors.
  - The third row has two type errors: SafetyInjection has the values Off and On, not False and True.
  - The second row violates two properties of condition tables - namely, Coverage (Overridden ∨ Overridden = Overridden ≠ *true*) and Disjointness (Overridden ∧ Overridden = Overridden ≠ *false*)

Table VII. Modified Table for Safety Injection

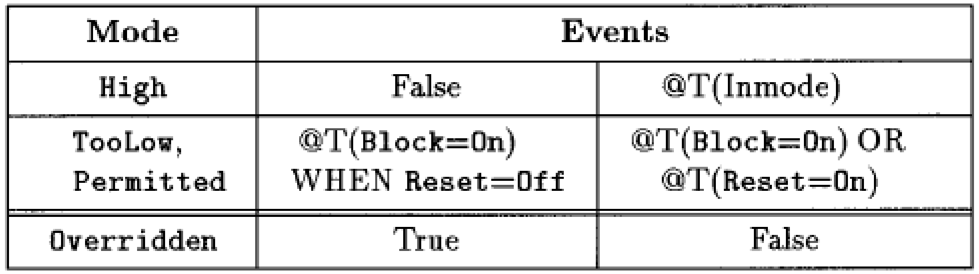


Mode	Conditions	
High, Permitted	True	False
TooLow	Overridden	Overridden
Safety Injection	False	True

4.1 Checking for Disjointness and Coverage

- Example. Table VIII is a variation of the event table in Table II. Running the consistency checker detects a Disjointness error in the second row of Table VIII.
  - In checking for Disjointness, the consistency checker evaluates the expression, [@T(Block=On) WHEN Reset=Off] ∧ [@T(Block=On) ∨ @T(Reset=On)] = *false*.
  - This expression is not a tautology. (→ Not disjointed)

Table VIII. Modified Table for Overridden



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

4.1 Checking for Disjointness and Coverage

$$
\begin{aligned}
&[@T(\mathsf{Block} = \mathsf{On})\ \mathrm{WHEN}\ \mathsf{Reset} = \mathsf{Off}] \wedge [@T(\mathsf{Block} = \mathsf{On}) \vee \\
&@T(\mathsf{Reset} = \mathsf{On})] \\
&\quad = \quad [@T(\mathsf{Block} = \mathsf{On})\ \mathrm{WHEN}\ \mathsf{Reset} = \mathsf{Off} \wedge @T(\mathsf{Block} = \mathsf{On})] \vee [@T(\mathsf{Block} = \\
&\quad\quad \mathsf{On})\ \mathrm{WHEN}\ \mathsf{Reset} = \mathsf{Off} \wedge @T(\mathsf{Reset} = \mathsf{On})] \qquad \text{(Distributive Law)} \\
&\quad = \quad [\mathsf{Block} = \mathsf{Off} \wedge \mathsf{Block}' = \mathsf{On} \wedge \mathsf{Reset} = \mathsf{Off} \wedge \mathsf{Block} = \mathsf{Off} \wedge \\
&\quad\quad \mathsf{Block}' = \mathsf{On}] \vee [\mathsf{Block} = \mathsf{Off} \wedge \mathsf{Block}' = \mathsf{On} \wedge \mathsf{Reset} = \mathsf{Off} \wedge \\
&\quad\quad \mathsf{Reset} = \mathsf{Off} \wedge \mathsf{Reset}' = \mathsf{On}] \qquad \text{(By (1))} \\
&\quad = \quad [\mathsf{Block} = \mathsf{Off} \wedge \mathsf{Block}' = \mathsf{On} \wedge \mathsf{Reset} = \mathsf{Off}] \vee \mathit{false} \\
&\qquad\qquad\qquad \text{(One-Input Assumption)} \\
&\quad = \quad \mathsf{Block} = \mathsf{Off} \wedge \mathsf{Block}' = \mathsf{On} \wedge \mathsf{Reset} = \mathsf{Off} \\
&\quad \neq \quad \mathit{false}
\end{aligned}
$$

- 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*.

4.1 Checking for Disjointness and Coverage

- Some checks, such as syntax and type checking, are straightforward.
- However, more complex are checks that depend on definitions, other than type definitions, in different parts of the specification or checks that require deductive reasoning.
- We have provided a semantic framework to reason formally about <u>assumptions</u>, 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 Bharadwaj [1996] for details).
  - For the general case, the tool may need user feedback to complete certain checks.

4.2 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.
  - Although tautology checking may have worst-case behavior that is exponential in the size of the expression [Garey and Johnson 1979], 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.

- Our experience with consistency checking is that the number of subproblems and the size of each subproblem grow 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

4.3 Prototype 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.

5. Applying Consistency Checks

5.1 Checks on Condition Tables

5.2 Checks on Mode Transition Tables

5.3 Manual vs. Automated Checks

6. Related Work

6.1 Decision Table Processors

6.2 Tablewise

6.3 RSML

6.4 Consistency Checking in Tablewise, RSML, and SCR

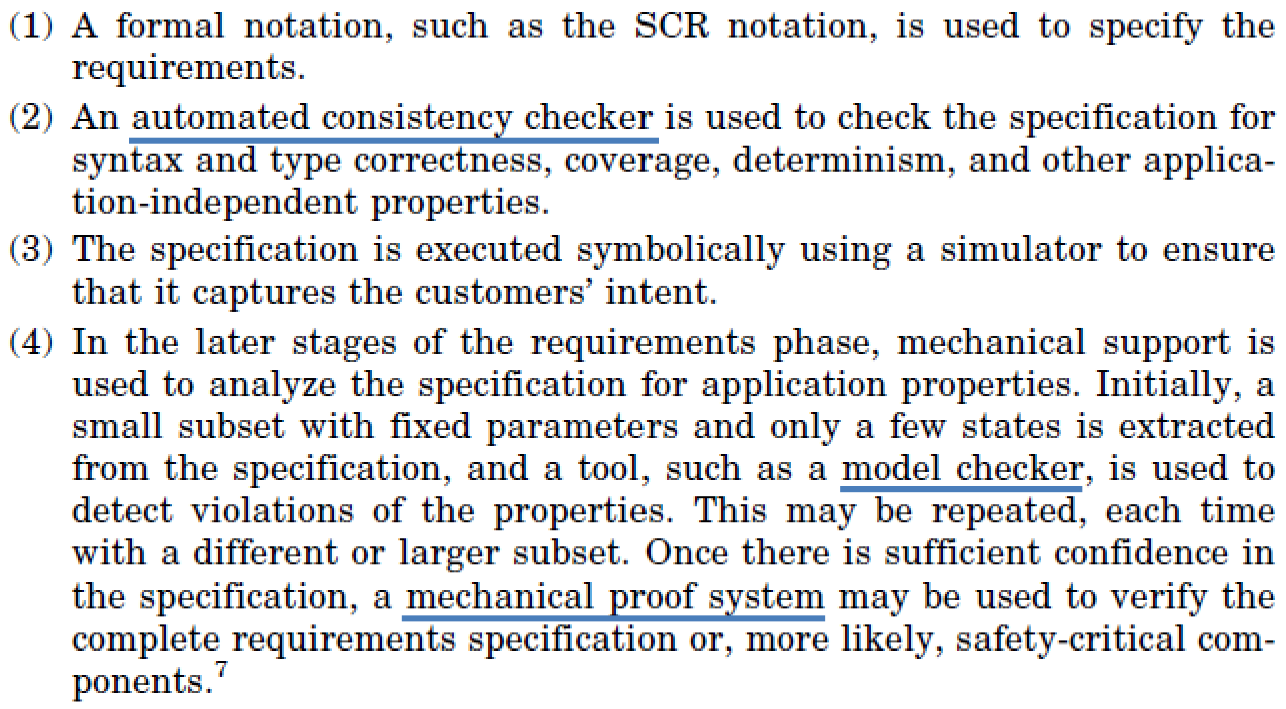
6.5 Mechanical Proof Systems

6.6 Model Checking

6.7 Detecting Errors by Inspection

7. Requirements Process

- We envision the following process for developing 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]

8. Concluding Remarks

- Tools for consistency checking can be highly effective for detecting errors in requirements specifications.
- 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.