Application of Tabular Methods to the Specification and Verification of a Nuclear Reactor Shutdown System

M. Lawford (lawford@mcmaster.ca)*
Dept. of Comp. and Software, McMaster University, Hamilton, ON, CANADA L8S 4L7

P. Froebel (peter.froebel@ontariopowergeneration.com)
G. Moum (g.moum@ontariopowergeneration.com)
Ontario Power Generation, 700 University Ave., Toronto, ON, CANADA M5G 1X6

August 20, 2001

Abstract. This paper describes the use of tabular methods at Ontario Power Generation Inc. (OPGI) [1] on the Darlington Nuclear Generating Station Shutdown System (SDS) Trip Computer Software Redesign Project. We first motivate the selection of tabular methods and provide an overview of the Systematic Design Verification (SDV) procedure. After reviewing some preliminary concepts, the paper describes how the Software Engineering Standards and Methods (SESM) Tool suite is used with SRI's automated proof assistant, PVS, to provide tool support for the use of tabular methods in the software engineering process. Examples based upon the Systematic Design Verification of an actual SDS subsystem are used to illustrate the benefits and limitations of the current implementation of the formal method. Finally, the paper discusses related work, draws conclusions regarding the effectiveness of the method and examines how its limitations can be addressed by further theoretical and applied work.

Keywords: Safety Critical Software, Tabular Methods, Formal Specification and Verification, Theorem Proving, PVS

1. Introduction

The main purpose of this paper is to provide a tutorial introduction to the application of tool supported tabular methods to the specification and verification of safety critical software. Emphasis is placed on how to make the method practical. The authors are of the opinion that it is important to make such formal methods accessible to practicing engineers to improve the quality of both safety critical software and formal methods. A secondary goal is to make formal methods theoreticians aware of some of the implementation issues that must be addressed to make formal methods widely applicable to industrial software applications. To achieve these goals we will examine the Systematic Design Verification (SDV) procedure that was applied as an integral part of the software development process

[1] Ontario Power Generation Inc. is the electricity generation company created from Ontario Hydro on April 1, 1999.

* Partially supported by the Natural Sciences and Engineering Research Council of Canada.

for the Darlington Nuclear Generating Station Shutdown System (SDS) Trip Computer Software Redesign Project. Henceforth we will refer to this project as the SDS Redesign Project.

Ontario Power Generation Inc. (OPGI) and Atomic Energy of Canada Limited (AECL) have jointly defined a detailed engineering standard to govern the specification, design and verification of safety critical software systems. The CANDU Computer Systems Engineering Centre of Excellence Standard for Software Engineering of Safety Critical Software [17] states the following as its first fundamental principle:

> *The required behavior of the software shall be documented using mathematical functions in a notation which has well defined syntax and semantics.*

The current implementation of the software engineering process makes extensive use of tool supported tabular methods. It results in the production of a coherent set of documents that allow for the static analysis of the properties of the requirements, described in the Software Requirements Specification (SRS), and then verifies the design, described in the Software Design Description (SDD), against the requirements.

While proponents of formal methods have been advocating their use in the development and verification of safety critical software for over two decades [26, 12, 28], there have been few full industrial applications utilizing rigorous mathematical techniques. This is in part due to industry's perception that formal methods are difficult to use and fail to scale to "real" problems. To address these concerns, a method must supply integrated tool support to automate much of the routine mechanical work required to perform formal specification, design and verification.

There have been some notable industrial and military applications of tool supported formal methods to software systems requirements analysis (e.g., [21, 7, 11, 3]), though typically the formal methods advocates were not given the opportunity to fully integrate their techniques with the overall software engineering process. As a result these applications required at least some reverse engineering of existing requirements documents into the chosen formalism. A potential problem of this scenario is that two "specifications" may result, the original, often informal, specification used by developers, and the formal specification used by verifiers. Rather than focusing on requirements analysis, the part of the software engineering process presented here focuses on verifying that a design meets the specified software requirements using documents that, from the start, have been designed for use with the tools of the formal method.

The SDS Redesign Project represents one of the first times that a software engineering process has been designed with the application of tool supported formal methods to specification and verification as a primary goal. This change in focus was necessitated by regulatory requirements, a situation that is becoming increasingly common for industries utilizing software in safety critical applications. The major factors considered in choosing a particular formal method for the Redesign Project were:

- learning curve and ease of understanding
- ability to provide tool support
- previous history indicating the ability to successfully scale to industrial applications

We now address these three points in more detail.

Since they are frequently used in many settings and provide important visual information, tables are easily understood by domain experts, developers, testers, reviewers and verifiers. Other methods such as VDM or Z utilize unfamiliar notation and special languages with a significant learning curve [35]. As we describe later, the Systematic Design Verification (SDV) procedure avoids this problem through the use of tabular notation in both the requirements and design documents utilized by all project team members. To create the tabular specifications, custom "light-weight" formal methods tools (in the sense of [6, 10]) are used to help create and debug the tables from within a standard word processor. To perform the verification these tools then extract the tables from the documents and generate input files for SRI's Prototype Verification System (PVS) automated proof assistant [33]. Further explanation of the tool support is provided in Section 3.4.

OPGI had strong evidence that a verification procedure employing tabular methods would meet the requirements of the Redesign Project. Prior to the Redesign Project, OPGI successfully used manual manipulation of tables with an earlier version of the same verification procedure detailed in Section 3 to verify a smaller scale Digital Trip Meter system [23]. The creation of the specialized tools that allowed verification to be done with the help of PVS played a large role in making the method feasible for the larger Redesign Project. A further reason for the adoption of tabular methods is that they have been successfully applied to a wide variety of applications. In particular, they have been used successfully with PVS on problems such as the verification of hardware division algorithms similar to the one that caused the Pentium floating point bug [31]. Further description of the tools support and rationale for OPGI's choice of tools is provided in Section 3.4.

Tabular methods are well suited to the documentation of the SDS control functions that typically partition the input domain into discrete modes or operating regions. The examples in Section 4 aptly demonstrate this property while illustrating some of the other major benefits of this, and other, tool supported formal methods, including providing:

- Independent checks which are unaffected by the verifier's expectations,
- Domain coverage - Tools can often be used to check *all* input cases - something that is not always possible or practical with testing,
- Detection of implicit assumptions and ambiguous/inconsistent specifications,
- Additional capabilities such as the generation of counter examples for debugging, type checking, verifying whole classes of systems, etc.

The examples used to illustrate these points are not contrived. They are based upon real examples from the SDS Redesign Project, although they have been simplified to eliminate unimportant details that would distract the reader from key concepts.

Section 2 provides an overview of the basic concepts required to understand the tabular methods examples from the SDS Redesign Project. Section 3 describes how a functional version of the 4-variable model of [30] can be decomposed to facilitate tool support and to reduce the manual effort required to perform and document the specification and verification tasks. It also describes the underlying semantics of our model and explains how the Software Engineering Standards and Methods (SESM) Tool suite is used with PVS to provide tool support for the use of tabular methods throughout the software engineering process and the Systematic Design Verification (SDV) procedure in particular. Examples based upon the verification of an actual SDS subsystem are employed in Section 4 to illustrate the benefits and limitations of the current implementation of the formal method and tools. Section 5 provides discussion of related work focusing on the application of tool supported formal methods to industrial control software design. Finally, in Section 6, the paper draws conclusions regarding the effectiveness of the method and examines how its limitations can be addressed by further theoretical and applied work.

2. Preliminaries

In this section we define the notation we will use throughout the remainder of the paper. We review the key principles of sequent calculus and tabular specifications that will allow the reader to interpret the examples and finally we provide a brief overview of the support for analysis of tabular specifications available in PVS.

2.1. Notation

Functions and relations are shown in italics (e.g., f, REQ). All sets of time series vectors from the 4-variable model are shown bold (e.g., $\mathbf{BM}$). All other mathematical terms are shown in italics (e.g., $bm \in \mathbf{BM}$).

For a set V_i, we will denote the *identity map on the set* V_i by id_{V_i} (i.e., $id_{V_i} : V_i \to V_i$ such that $v_i \mapsto v_i$). Given functions $f : V_1 \to V_2$ and $g : V_2 \to V_3$, we will use $g \circ f$ to denote *functional composition* (i.e., $g \circ f(v_1) = g(f(v_1))$). The *cross product* of functions $f : V_1 \to V_2$ and $f' : V_1' \to V_2'$, defines a function $f \times f' : V_1 \times V_1' \to V_2 \times V_2'$ such that $(v, v') \overset{f \times f'}{\mapsto} (f(v), f'(v'))$.

Denote the set of all equivalence relations on V by $Eq(V)$. Any function $f : V \to R$ induces an equivalence relation $\ker(f) \in Eq(V)$, the *equivalence kernel* of f, given by $(v_1, v_2) \in \ker(f)$ if and only if $f(v_1) = f(v_2)$. We define the standard partial order on equivalence relations as follows. Given equivalence relations $\theta_1, \theta_2 \in Eq(V)$, we say that θ_1 *is a refinement of* θ_2, written $\theta_1 \le \theta_2$, iff $(v, v') \in \theta_1$ implies $(v, v') \in \theta_2$ for all $(v, v') \in V \times V$. We can now formally state a basic existence claim for functions that will be used later in Section 4.2 when we discover that a functional specification is unimplementable as stated.

Claim 1. Given two functions with the same domain, $f : V_1 \to V_3$ and $g : V_1 \to V_2$, there exists $h : V_2 \to V_3$ such that $f = h \circ g$ iff $\ker(g) \le \ker(f)$.

2.2. The PVS Sequent Calculus

In this subsection we define the logical notation we will use and provide a brief overview of the properties of the PVS sequent calculus that are required to interpret the examples of Section 4. The reader is referred to [33] for a more detailed introduction to PVS.

Let P_i, $i = 1, \ldots n$ and Q_j, $j = 1, \ldots, m$ be formulas in higher order logic and let $\vdash$ denote syntactic entailment. Henceforth we will use $\neg P_1$, $P_1 \wedge P_2$ and $Q_1 \vee Q_2$ to denote negation, conjunction and disjunction respectively. We will use $P_1 \Rightarrow Q_1$ as an abbreviation for $\neg P_1 \vee Q_1$ while the special symbols $\top$ and $\bot$ will be used to denote $TRUE$ and $FALSE$. To reduce the number of parentheses required to write our formulas we assume the following decreasing order of precedence of operations: $\neg, \wedge, \vee, \Rightarrow$.

In general when trying to prove properties of software, we will assume properties regarding the system inputs are all true (the P_i's), and try to prove one or more properties regarding the output (one or more Q_j's) is true. We formally write, $P_1, P_2, \ldots, P_n \vdash Q_1 \vee Q_2 \vee \ldots \vee Q_m$, or equivalently $P_1 \wedge P_2 \wedge \ldots \wedge P_n \vdash Q_1 \vee Q_2 \vee \ldots \vee Q_m$.

In sequent calculus this is written as:

$$\begin{array}{|l} P_1 \wedge P_2 \wedge \ldots \wedge P_n \\ \hline Q_1 \vee Q_2 \vee \ldots \vee Q_m \end{array} \quad \text{or equivalently} \quad \begin{array}{|l} P_1 \\ P_2 \\ \vdots \\ P_n \\ \hline Q_1 \\ Q_2 \\ \vdots \\ Q_m \end{array} \tag{1}$$

In the second "sequent", there are implicit $\wedge$'s between the premises and implicit $\vee$'s between the conclusions. We will use sequent and equivalent standard logical notation interchangeably throughout the paper.

Definition 1. For the sequents in (1) the *characteristic formula* is $P_1 \wedge \ldots P_n \Rightarrow Q_1 \vee \ldots Q_m$.

Note that the characteristic formula is a logical theorem iff the sequent is provable.

Below are two special cases of sequents:

$$\text{(i)} \begin{array}{|l} \\ \hline Q_1 \\ \vdots \\ Q_m \end{array} \qquad \text{(ii)} \begin{array}{|l} P_1 \\ \vdots \\ P_n \\ \hline \\ \end{array} \tag{2}$$

Case (2)(i), when there are no premises, corresponds to showing that the disjunction of the conclusions is a logical theorem ($\vdash Q_1 \vee \ldots \vee Q_m$), while (2)(ii), when there are no conclusions, corresponds to proving that the premises are inconsistent ($P_1 \wedge \ldots \wedge P_n \vdash \bot$).

2.2.1. *Proofs in PVS Sequent Calculus*

Proofs are done by transforming the sequent until one of the following forms is obtained:

$$\begin{array}{|l} \vdots \\ P \\ \\ \hline P \\ \vdots \end{array} \quad \text{or} \quad \begin{array}{|l} \vdots \\ \\ \hline \top \\ \vdots \end{array} \quad \text{or} \quad \begin{array}{|l} \vdots \\ \bot \\ \hline \vdots \end{array} \tag{3}$$

The sequent transformations used in sequent calculus proofs can all be translated to proof rules in natural deduction or other axiomatizations of logic and vice versa. The following are sequent transformations that we will make use of in the examples.

Negations occurring at the top level in premises of conclusion can be eliminated (or added) by moving the formula to the other side of the sequent and dropping (adding) the negation:

$$\text{(i)}\begin{array}{|c} P_1 \\ \hline \neg Q \\ Q_1 \\ Q_2 \end{array} \iff \begin{array}{|c} P_1 \\ Q \\ \hline Q_1 \\ Q_2 \end{array} \qquad \text{(ii)}\begin{array}{|c} P_1 \\ \neg P \\ \hline Q_1 \\ Q_2 \end{array} \iff \begin{array}{|c} P_1 \\ \hline P \\ Q_1 \\ Q_2 \end{array} \tag{4}$$

The above sequent transformation rules are implemented in PVS as part of such commands as FLATTEN.

When using sequent calculus, it is common to split a proof into smaller proofs or *subgoals*. This can be done by the two transformations shown in (5) below:

$$\text{(i)}\quad \begin{array}{ccc} & \begin{array}{|c} \vdots \\ \hline Q_1 \wedge Q_2 \\ \vdots \end{array} & \\ \swarrow & & \searrow \\ \begin{array}{|c} \vdots \\ \hline Q_1 \\ \vdots \end{array} & & \begin{array}{|c} \vdots \\ \hline Q_2 \\ \vdots \end{array} \end{array} \qquad \text{(ii)}\quad \begin{array}{ccc} & \begin{array}{|c} P_1 \vee P_2 \\ \vdots \\ \hline \vdots \end{array} & \\ \swarrow & & \searrow \\ \begin{array}{|c} P_1 \\ \vdots \\ \hline \vdots \end{array} & & \begin{array}{|c} P_2 \\ \vdots \\ \hline \vdots \end{array} \end{array} \tag{5}$$

The transformation (5)(i) uses the fact that, $P_1, \ldots, P_n \vdash Q_1 \wedge Q_2$ iff $P_1, \ldots, P_n \vdash Q_1$ and $P_1, \ldots, P_n \vdash Q_2$, to "split" $\wedge$ in the conclusions into two subproofs. Similarly it is possible to split $\vee$ in the premises into two subgoals since $P_1 \vee P_2 \vdash Q$ iff $P_1 \vdash Q$ and $P_2 \vdash Q$

The splitting transformations are implemented by the low level PVS command SPLIT. As mentioned above, these and other transformations, are used to transform the sequent and all of its subgoals into one of the forms in (3). While typically many of these low level commands are combined into a single high level proof step or "strategy" in PVS, a thorough understanding of these low level sequent transformations is useful for understanding how PVS proofs work and what it means when they fail. We now consider how it may be possible to interpret a sequent when it is not possible to reach one of the final forms in (3).

2.2.2. *Unprovable Sequents and Counter Examples*

Suppose we wanted to use sequent calculus to check if the following formula is a logical theorem:

$$(Q \Rightarrow P_1 \vee P_2) \wedge P_1 \wedge (P_2 \Rightarrow Q) \Rightarrow Q$$

Using the fact that $(P_i \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$ and applying in sequence the transformations (1), (4)(i), and (1), we obtain the sequent:

$$\begin{array}{|l} Q \Rightarrow P_1 \vee P_2 \\ P_1 \\ P_2 \Rightarrow Q \\ \hline Q \end{array}$$

After spitting disjunctions and implications in the premises and further application of the transformations, we obtain the (unprovable) sequent:

$$\begin{array}{|l} P_1 \\ \hline Q \\ P_2 \end{array}$$

which has characteristic formula $P_1 \Rightarrow (Q \vee P_2)$. This formula is false when $P_1 = \top$ and $P_2 = Q = \bot$. One can easily verify that this assignment provides a counter example showing the original formula is not a logical theorem.

2.3. Tabular Specification of Functions

The example below and the description of PVS and its support for tables in the following section are largely based upon [25].

A function $f : T_1 \times \ldots \times T_m \rightarrow T_r$ may have a tabular representation:

$$f(x_1, \ldots, x_m) = \begin{array}{|c|c|c|c|} \hline c_1 & c_2 & \ldots & c_n \\ \hline e_1 & e_2 & \ldots & e_n \\ \hline \end{array} \quad \text{or} \quad \begin{array}{|c||c|} \hline c_1 & e_1 \\ \hline c_2 & e_2 \\ \hline \vdots & \vdots \\ \hline c_n & e_n \\ \hline \end{array} \tag{6}$$

Here each c_i is a boolean expression and e_i is a term of type T_r. The interpretation is that when c_i is true f returns e_i. In this case for the table to properly define a (total) function, it is sufficient for it to satisfy the following two conditions:

Disjointness: requires that the columns (rows) do not overlap. i.e., $i \neq j \Rightarrow \neg(c_i \wedge c_j)$.

Completeness: requires that at least one column (row) is applicable to every input. i.e., $(c_1 \vee c_2 \vee \ldots \vee c_n)$ is always $TRUE$.

The disjointness condition can be weakened to make the conditions both necessary and sufficient by requiring that where there is overlap between columns, the columns produce the same results. In practice, such overlaps may cause problems. It is often the case that mathematically equivalent expressions are not computationally equivalent due to, e.g., numerical errors. This opens up the possibility of different outcomes for the same specification depending upon how the designer implements the specified function.

Example 1. Let x be a real valued variable. Then the function:

$$sign(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

has the equivalent tabular representation:

$x < 0$	$x = 0$	$x > 0$
-1	0	1

For the purposes of this paper we will restrict ourselves to simple horizontal and vertical condition tables or minor variations thereof such as the ones shown above. More compact tabular representations such as Structured Decision Tables [23], Normal, Inverted and Vector Function tables [27] exist for the specification of complex functions. The fundamental notions of disjointness and completeness are easily generalized to these other types of tables.

2.4. PVS Support for Tables

PVS consists of a specification language for creating input files containing user defined theories and an interactive theorem prover and decision procedures for typechecking and verifying these theories. The specification language is a higher order logic based on the simple theory of types augmented by dependent types and predicate subtypes. Although the specification language allows for the addition of axioms to the system, their use is discouraged since any additional axioms may introduce inconsistencies into the proof system, weakening any guarantees of the correctness of the results. If the specification contains no additional axioms, then typechecking can guarantee that the system introduces no additional inconsistencies [32]. The system provides strong assurances that definitional constructs such as recursive function definitions are conservative extensions of the logic.

While much of the typechecking required to ensure conservative extension of the PVS logic can be done automatically, predicate subtypes and, as we will see, tabular specification of functions, can cause PVS to generate proof obligations called Type Correctness Conditions (TCCs) that must be discharged using theorem proving. The proof strategies built into the theorem prover automatically handle many of these proof obligations, leaving the user to interactively prove the more complex TCCs. The proof of any theorems in a user input file are considered incomplete until the user defined theory and any theories it imports have been typechecked and any generated TCCs have been proved.

The PVS specification language provides facilities for declaring types, functions, variables, constants and formulas. It also provides the COND construct as a basic method of specifying function tables. The COND construct for the

`COND`	
c_1 `->` e_1`,`	`IF` c_1 `THEN` e_1
c_2 `->` e_2`,`	`ELSIF` c_2 `THEN` e_2
$\vdots$	$\vdots$
c_{n-1} `->` e_{n-1}`,`	`ELSIF` c_{n-1} `THEN` e_{n-1}
c_n `->` e_n	`ELSE` e_n
`ENDCOND`	`ENDIF`

Figure 1. General COND construct and PVS interpretation

general table of (6) is shown on the left side of Figure 1. The right side shows the equivalent IF-THEN-ELSE statements that PVS uses as the internal interpretation of the COND statement. The result is that the standard built in PVS commands easily handle the COND construct by first translating them into the IF-THEN-ELSE statments.

2.4.1. *Typechecking COND Statements*

The following PVS statements uses the PVS COND construct to define the *sign* function of Example 1.

```
signs: TYPE = { i: int | i >= -1 & i <= 1}

  sign_cond(x:real): signs =
    COND
      x<0 -> -1,
      x=0 -> 0,
      x>0 -> 1
    ENDCOND
```

The & above represents conjunction in the PVS specification language. Use of COND causes PVS to automatically generate Disjointness and Completeness TCCs (proof obligations). These can often be automatically discharged (proved) by PVS' built in proof strategies. As we will see in Section 4, when the built in proof strategies fail, the resulting unprovable sequent(s) can often provide useful information regarding the incompleteness or inconsistency of specifications.

```
% Disjointness TCC generated for
 % COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  % unfinished
sign_cond_TCC3: OBLIGATION
```

```
     (FORALL (x: int):
 NOT (x < 0 AND x = 0)
 AND NOT (x < 0 AND x > 0)
 AND NOT (x = 0 AND x > 0));


% Coverage TCC generated for
% COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  % unfinished
sign_cond_TCC4: OBLIGATION
(FORALL (x: int): x < 0 OR x = 0 OR x > 0);
```

2.4.2. *PVS Table Construct*

The PVS specification language provides various TABLE constructs to make the prover input more readable. For example:

```
  sign_htable(x:real): signs = TABLE
                 %-------------------%
                 |[ x<0 | x=0 | x>0 ]|
                 %-------------------%
                 |  -1  |  0  |  1  ||
                 %-------------------%
                 ENDTABLE
```

The TABLE constructs are translated into PVS COND constructs for typechecking and proving purposes. It is possible to represent more complex tables such as two dimensional tables through the use of nested CONDs.

3. Systematic Design Verification Procedure and Tools

This section provides an overview of the Systematic Design Verification (SDV) procedure and corresponding tool support employed on the Redesign Project. We highlight elements of the process, such as the decomposition of proof obligations, that facilitate tool support and reduce the effort required to perform rigorous design verification, including creation and maintenance of the process documents. Although the SDV procedure covers other types of verification problems, such as verification of pseudocode, real-time properties, input queues, etc., we will concentrate on the verification of functional properties utilizing tabular notation. The reader is referred to [24] for the complete procedure.

3.1. SDV Procedure Overview

The software engineering process described here is based upon the *Standard for Software Engineering of Safety Critical Software* [17] that was jointly developed by OPGI and AECL. This standard requires that the software development and verification be broken down into series of tasks that result in the production of detailed documents at each stage. The software development stages relevant to this paper are governed by the Software Requirements Specification Procedure [16] and the Software Design Description Procedure [22]. These procedures respectively produce the Software Requirements Specification (SRS) and Software Design Description (SDD) documents. In addition to other methods, these documents make use of a form of Parnas' tabular representations of mathematical functions [15, 27] to specify the software's behavior. Tables provide a mathematically precise notation (see [14] for the formal semantics) for the SRS and SDD in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike [35].

The underlying models of both the SRS and SDD are based upon Finite State Machines (FSM). The SDD adds to the SRS functionality the scheduling, maintainability, resource allocation, error handling, and implementation dependencies. The specification technique for defining the implementation is based upon an abstract state machine model that will execute the implemented source code. The primary difference between this abstract state machine and the FSM describing the SRS is that execution is not instantaneous, but takes a finite amount of time, and thus the order of execution must be specified to avoid race conditions. The SRS is produced by software experts with the help of domain experts. It is used by lead software developers to produce the SDD which is then used by all the developers to produce the actual source code.

The software engineering standard [17] requires that the SDD be formally verified against the SRS and then the code formally verified against the SDD to ensure that the implementation meets the requirements. These formal verifications are governed by the SDV Procedure and Systematic Code Verification Procedure. For the purposes of this paper we will concentrate on the SDV process.

The objective of SDV is to verify, using mathematical techniques or rigorous arguments, that the behavior of every output defined in the SDD is in compliance with the requirements for the behavior of that output as specified in the SRS. It is based upon a specialization of the 4-variable model of [30] that verifies the functional equivalence of the SRS and SDD by comparing their respective one step transition functions. The resulting proof obligation in this special case:

$$REQ = OUT \circ SOF \circ IN \tag{7}$$

is illustrated in the commutative diagram of Figure 2. Here REQ represents the SRS state transition function mapping the monitored variables $\mathbf{M}$ to the controlled variables $\mathbf{C}$. The function SOF represents the SDD state transition

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "REQ"] \arrow[d, "IN"'] & \mathbf{C} \\
\mathbf{I} \arrow[r, "SOF"] & \mathbf{O} \arrow[u, "OUT"']
\end{tikzcd}$$

Figure 2. Commutative diagram for 4-variable model

function mapping the behavior of the implementation input variables represented by statespace $\mathbf{I}$ to the behavior of the software output variables represented by the statespace $\mathbf{O}$. The mapping IN relates the specification's monitored variables to the implementation's input variables while the mapping OUT relates the implementation's output variables to the specification's controlled variables. The following section briefly outlines the refinement of the relational methods in [30] to the simple functional case in (7).

3.2. Specialization of the 4-Variable Model

In the 4-variable model of [30], each of the 4 "variable" state spaces $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set of functions of a single real valued argument that return a vector of values - one value for each of the quantities or "variables" associated with a particular dimension of the statespace. For instance, assuming that there are n_M monitored quantities, which we represent by the variables $m_1, m_2, \ldots, m_{n_M}$, then, the possible timed behavior of the variable m_i can be represented as a function $m_i^t : \mathbb{R} \to \mathit{Type}(m_i)$, where $m_i^t(x)$ is the value of the quantity m_i at time x. We can then take $\mathbf{M}$ to be the set of all functions of the form $m^t(x) = (m_1^t(x), m_2^t(x), \ldots, m_{n_M}^t(x))$. Thus the relations corresponding to the arrows of the commutative diagram then relate vectors of functions of a single real valued argument.

In order to simplify the 4-variable model to a FSM setting, we restrict ourselves to the case where each of the 4 "variables" $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set of "time series vectors". For example, $\mathbf{M}$ actually refers to all possible sets of observations ordered (and equally spaced) in time, each observation being a vector of n_M values. We will use the term *monitored variable* to refer to quantity m_i which is the ith element in the vector ($i \in \{1, \ldots, n_M\}$). Let $m \in \mathbf{M}$ be a time series vector of observations of the monitored variables. With a slight abuse of notation, we will use $m_i(z)$ to denote the zth observation of the ith element ($z \in \{0, 1, 2, \ldots\}$) of the monitored variables for the time series vector m. Similarly $m(z)$ represents the zth observation of the n_M values in the monitored variable vector for time series m.

For this model, the time increment between each of the observations is defined to be the positive real value $\delta > 0$. Thus observation z corresponds to time $(z*\delta)$. The increment δ is taken to be at least an order of magnitude less than any time measurements of interest. The value of m_i at any point between two observations (i.e., in the range of time $[z*\delta, (z+1)*\delta)$) is defined to be equal to $m_i(z)$.

Each of the "variables" $\mathbf{M}, \mathbf{C}, \mathbf{I}, \mathbf{O}$ in the specialized 4-variable model has the same frequency of observation, but may have a different number of values in its vector. The value n_M is defined to be the number of elements in $\mathbf{M}$, which are observed over time, while n_I is defined to be the number of elements in $\mathbf{I}$, which are observed over time. Normally $n_I = n_M$. Similarly n_O is defined to be the number of elements in $\mathbf{O}$, which are observed over time and n_C is defined to be the number of elements in $\mathbf{C}$, which are observed over time. Normally, $n_C = n_O$.

Requirements (REQ): The required behavior of the subsystem is described with REQ. At OPGI REQ is modeled as a FSM, defining a relation over $\mathbf{M} \times \mathbf{C}$. While, in general, the FSM could be nondeterministic, much of the system behavior can be modeled by a deterministic FSM with the result that for the verification of these properties we can assume that REQ is a function (i.e., $REQ : \mathbf{M} \to \mathbf{C}$).

In this case a new set of time series vectors, $\mathbf{S}$, is introduced to describe the state of the FSM. Let $c \in \mathbf{C}$, $m \in \mathbf{M}$, $s \in \mathbf{S}$, and $z \in \{0, 1, 2, \ldots\}$. The zth value of a controlled variable time series vector $c(z)$ depends on both the values of $m(z)$ and $s(z)$, related by the vector function $OUTPUT$ (i.e., $c(z) = OUTPUT(m(z), s(z))$). Also, the value $s(z+1)$ depends on both the values of $m(z)$ and $s(z)$, related by the vector function $NEXTSTATE$. (i.e., $s(z+1) = NEXTSTATE(m(z), s(z))$).

The SRS procedure [16] shows how a set of functions $f_1, f_2, \ldots f_j$ can be defined such that when a subset of them are composed, they define the OUTPUT function. When a different, though not necessarily disjoint, set of them are composed, they define the $NEXTSTATE$ function. We have called the process of defining these functions the "decomposition of REQ".

Design (SOF): The implemented behavior of the subsystem is described with SOF. SOF can be modeled as a directed graph with $p+2$ nodes. Within this graph, each node is either one of p FSMs, or $\mathbf{I}$, or $\mathbf{O}$, and each edge represents data flow between two of these. The node containing $\mathbf{I}$ must not be the destination of any edge. The node containing $\mathbf{O}$ must not be the source of any edge. In this way, SOF defines a (functional) relation over $\mathbf{I} \times \mathbf{O}$. If the design is produced following the SDD procedure, then each of the FSMs represents a program called from the mainline. We assume a constant mainline loop structure, with each program called 1 or more times within the loop.

For a large number of the implementation properties, the FSMs composing SOF can be modeled as deterministic FSM allowing us to consider the special case when SOF defines a function. In this case, when both REQ and SOF are functions, if we are also able to restrict ourselves to functional maps for IN and OUT, we can verify the commutative diagram in Figure 2 by comparing the one step transition functions of the FSMs defining REQ and SOF. More detailed descriptions of the underlying SRS and SDD models can be found in [16] and [22], respectively, as well as [24].

3.3. Decomposing the Proof Obligations

In Figure 3 we decompose the proof obligation (7) to isolate the verification of hardware interfaces. The $\mathbf{M}_p$ and $\mathbf{C}_p$ state spaces are the software's internal rep-

$$\begin{tikzcd}
\mathbf{M} \arrow[rrr, "REQ"] \arrow[d, "IN"'] \arrow[dr, "Abst_M"] & & & \mathbf{C} \arrow[dl, "Abst_C"'] \\
\mathbf{I} \arrow[r, "SOF_{in}"] & \mathbf{M}_p \arrow[r, "SOF_{req}"] & \mathbf{C}_p \arrow[r, "SOF_{out}"] & \mathbf{O} \arrow[u, "OUT"']
\end{tikzcd}$$

Figure 3. Vertical decomposition: Isolation of hardware hiding proof obligations

resentation of the monitored and controlled variables, referred to as the *pseudo-monitored* and *pseudo-controlled variables*, respectively. The proof obligations associated with SDV then become

$$Abst_C \circ REQ = SOF_{req} \circ Abst_M \tag{8}$$
$$Abst_M = SOF_{in} \circ IN \tag{9}$$
$$id_{\mathbf{C}} = OUT \circ SOF_{out} \circ Abst_C \ . \tag{10}$$

The first of these equations represents a comparison of the functionality of the system and should contain most of the complexity of the system. The last two represent comparisons of the hardware hiding software of the system. These obligations are often fairly straightforward and are discharged manually.

As an example to help the reader interpret the above decomposition, suppose an actual physical monitored plant parameter belonging to $\mathbf{M}$ is the temperature of the primary heat transport system which might have a current value of 500.3 Kelvin. The hardware corresponding to the temperature sensors and A/D converters might map this via IN to a value of 3.4 volts in a parameter in $\mathbf{I}$. A hardware hiding module might then process this input corresponding to the mapping SOF_{in}, producing a value of 500 Kelvin in the appropriate temperature variable belonging to the software state space $\mathbf{M}_p$. Further "vertical" decomposition is performed by isolating outputs and in effect restricting $\mathbf{M}$ and projecting $\mathbf{C}$ to the variables relevant to a particular subsystem such as the pressure sensor trip described in the Section 4.2.

The observant reader may have noted that the controlled variable abstraction function is defined as $Abst_C : \mathbf{C} \rightarrow \mathbf{C}_p$ which is seemingly the "wrong" direction. The proof obligation (10) forces $Abst_C$ to be invertible, preventing the possibility of trivial designs for SOF_{req} being used to satisfy the main obligation (8). As we will see below, this allows the verifier to define only one abstraction mapping for each pair of corresponding SRS and SDD state variables that occur as both inputs and outputs in the decomposition. The SDV procedure provides recourse for the case when there is not a 1-1 correspondence between $\mathbf{C}$ and $\mathbf{C}_p$ through the use of a pseudo-SRS that can be defined to more closely match the SDD. The interested reader is referred to [23] for further details.

Typically the verification of a subsystem as represented by (8), the inner part of the commutative diagram, can be decomposed "horizontally" at both the SRS and SDD level into a sequence of intermediate verification steps, thereby reducing the larger, more complex proof obligation into a number of smaller, more manageable verification tasks. This is represented in Figure 4 where each equality of the form:

$$SOF_i \circ Abst_{V_{i-1}} = Abst_{V_i} \circ REQ_i \tag{11}$$

becomes a verification block. Here $\mathbf{V}_i$ and $\mathbf{V}_{ip}$ are the statespaces associated with subsets of internal state variables that make up the abstract state machines (e.g., previous values of inputs, outputs or other internal state information relating to operating history). The price paid for this vertical and horizontal decomposition is

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "REQ_1"] \arrow[d, "Abst_M"'] \arrow[rrrrr, bend left, "REQ"] & \mathbf{V}_1 \arrow[r, "REQ_2"] \arrow[d, "Abst_{V_1}"'] & \mathbf{V}_2 \arrow[d, "Abst_{V_2}"'] & \ldots & \mathbf{V}_{n-1} \arrow[r, "REQ_n"] \arrow[d, "Abst_{V_{n-1}}"] & \mathbf{C} \arrow[d, "Abst_C"] \\
\mathbf{M}_p \arrow[r, "SOF_1"] \arrow[rrrrr, bend right, "SOF_{req}"'] & \mathbf{V}_{1p} \arrow[r, "SOF_2"] & \mathbf{V}_{2p} & \ldots & \mathbf{V}_{(n-1)p} \arrow[r, "SOF_n"] & \mathbf{C}_p
\end{tikzcd}$$

Figure 4. Horizontal (sequential) decomposition of proof obligations

that for each block the verifier must provide a cross reference between the internal variables making up the $\mathbf{V}_{i-1}, \mathbf{V}_i$ state spaces at the SRS level and the internal variables making up the $\mathbf{V}_{(i-1)p}, \mathbf{V}_{ip}$ state spaces at SDD level, as well as defining the abstraction functions, $Abst_{V_{i-1}}$ and $Abst_{V_i}$. Now the benefits of defining all the abstraction functions, including $Abst_C$, from top to bottom (SRS to SDD) in Figures 3 and 4 becomes more apparent. The values of many of the controlled variables from the previous execution pass of the SRS and SDD often become inputs to the calculation of current internal state and output variables. Similarly, state variables that are the output of one sequential block become the input of the following block. Defining all abstraction functions from top to bottom and then only performing the check for invertibility at the outputs embodied by (10) allows the verifier to use the same abstraction functions whether a state variable occurs at the input or output of a block. This technique reduces the number of abstraction functions required by up to one half.

3.4. Tool Support

An experience report of the first use of the above method prior to the use of support tools is detailed in [34]. The report cites the excessive amount of time required to perform the verification by hand as a major short fall of the method. As a result, OPGI and AECL undertook efforts to automate the SDV procedure.

The automation involved the development of a series of "light-weight" CASE tools known as the SESM Tools integrated with the PVS proof assistant from SRI. While the use of light-weight tools for the creation and analysis of requirements has been widely advocated in the literature (e.g., [6, 10]), combining lightweight tools with model-checkers and theorem provers can provide additional analysis and verification capabilities (e.g., [2, 11]). The light-weight tools such as those belonging to the SESM tool suite provide the ability to rapidly debug specifications and analyze simple properties, while a system such as PVS can be used for more in depth analysis and verification.

The SESM Tools have been designed to allow the designers and verifiers to use standard word processors such as Corel WordPerfect or Microsoft Word to create input documents employing tabular definitions of functions. This capability provides the team members with a familiar environment that results in highly readable software documentation. Figure 5 provides a graphic overview of the relationship between the documents and tools employed in the verification process. The word processor, augmented with the SESM tool macros, is used to

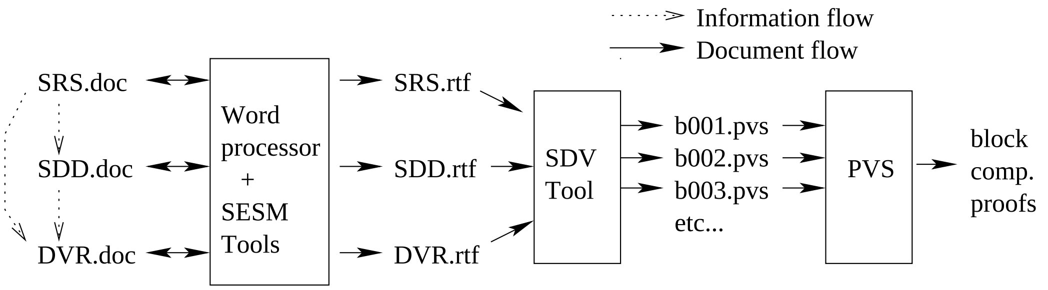

Figure 5. Relationship between tools and documents of the SDV process

create and debug, first the software requirements in the SRS, then the software design in the SDD and finally the Design Verification Report (DVR). The SESM tools provide completeness and consistency checks of the SRS, SDD and DVR documents that can be run offline on an entire document or invoked interactively via a macro from within the word processor to debug individual tabular function definitions as they are created. Additionally the tools check for the existence of circular dependencies in the requirements specified by the SRS.

The DVR provides the cross reference between the SRS and SDD inputs, outputs and functions and defines the abstraction functions that are part of the block decomposition of the proof obligations. These parts of the DVR are manually generated by the verifier with guidance provided by the SRS and SDD documents. Next, the word processor is used to create Rich Text Format (RTF) versions of these three documents that become input for the SDV Tool. RTF provides a standard input format for the SESM tools independent of the word processor used to create the documents. In addition to creating a log file with details of numerous document checks, the SDV tool also produces a PVS input file for each verification block containing the theorems, and associated function and type definitions, that must be proved for the block as required by the SDV procedure.

PVS is then used to typecheck the SESM tool output and prove the theorems. Type-checking of the tabular function definitions within PVS provides redundant verification of the completeness and and consistency checks and handle cases with more complicated types currently unsupported by the SESM tools.

Although the PVS specification language and interactive proof environment have their own steep learning curve, the verifiers require only a small subset of PVS' capabilities to perform the verification. Additionally, by designing the SESM tools to employ standard word processors for document creation, we have insured that no other team members require knowledge of the underlying proof system. While the examples presented in this paper make use of only a fraction of PVS' capabilities, integrating the SESM tools with PVS provides the opportunity to increase the scope of the computer assisted verification to include the real-time properties [1, 4, 20] and functional properties involving tolerances [19]. Additional reasons for choosing PVS were its existing support for tabular methods [25] integrated with theorem proving and model-checking and the abilities of its extensive type-checking capabilities to be used to detect software errors [32]. Notwithstanding these strong arguments in favor of PVS, the SDV tool has been designed so that after parsing the RTF documents it produces an intermediate flat text file format containing the relevant information prior to translation into PVS. This provides the ability to use alternative or supplemental verifications systems in the SDV process with relatively little effort.

The tools and procedures described here have been applied successfully to the SDS Redesign Project, which was completed early 1999. The project consisted of a complete redesign the software for two different trip computer systems. The complete systems are relatively small. Excluding comments and blank lines, one consists of approximately 12,000 lines of FORTRAN and assembler while the other was roughly 17,000 lines of Pascal and assembler. For both of the systems the SRS and SDD documents consisted mainly of formal tabular specifications and some informal description. Each requirements document (SRS) was approximately 400 pages while each design document (SDD) was over 500 pages. The resulting design verification documents (DVR) were each over 600 pages once completed (excluding PVS input and output), though this also includes the results of the verification procedure.

Much of the time and effort in the project was spent on document preparation. These documents form part of the formal submission required by the regulator and hence had to be prepared as part of the software engineering process employed on the SDS Redesign Project. The generation of the PVS code from documents took several hours on a Windows NT based 75 MHz Pentium system with 32MB of RAM and resulted in 11,000+ lines in 60 files for the first system and 13,000+ lines in 102 files for the second (line counts exclude blank lines and comments). The verification was performed by engineers with no previous experience with PVS or similar proof systems who received week-long training courses in PVS. The block comparison proofs and documentation of any discrepancies uncovered in the process took one person less than 2 weeks for each of the systems.

As the SESM tools are further refined and the verifiers gain more experience, this part of the SDV procedure should require less time. Also, with the ability to rerun proofs in batch mode, it should be possible to perform the formal verification of minor revisions much faster. The SDV procedure and SESM tools are now being used on the first revisions of the trip computer software. As a result of the success of these projects, OPGI is also considering expanding the use of the tools to the engineering of non-safety critical software systems.

4. Examples and Discussion

The Darlington Nuclear Generating Station Shutdown Systems (SDS) are “poised” systems that are not called upon to operate in normal conditions but rather monitor the plant parameters and react to shutdown or “trip” the reactor only if anomalous behavior is observed. The reactor process control is performed by a separate Digital Control Computer so that the safety critical shutdown functionality can be isolated in a separate high reliability system. This limits the scope of the formal verification activity to the smaller, more manageable SDS Trip Computer software.

This section demonstrates how tabular methods can be used with the SDV procedure of Section 3 to verify parts of a simplified reactor pressure trip subsystem. The examples have been simplified to highlight the main concepts and are formatted for clarity of presentation. For example, while a typical trip subsystem monitors plant parameters, (e.g., pressure and power) using multiple sensors, we have simplified the presentation to single sensors for each plant parameter. Although simplified, these examples are typical of many of the verification blocks from the SDS Redesign Project.

The examples deal with the power conditioning and sensor trip sections of a typical parameter trip subsystem. In the first example we see how proper application of tabular methods forces a designer to properly document all assumptions. The second example uses a simplified sensor trip to demonstrate how the verification task can be partitioned, and highlights the limitations of the current SDV tool regarding support for tolerances. The final example illustrates the benefits of the domain coverage provided by quantifier reasoning by discovering counter examples that may have been more difficult to detect using testing. Further limitations of the tool regarding the verification of timing properties are also discussed. In all the examples, the SDD tables utilize variable and function names of six characters or less since they are taken from the design targeted to a legacy FORTRAN compiler.

4.1. Detection of Implicit Assumptions

In this example we consider the design of the power conditioning of the subsystem. The reactor protective system is designed to provide coverage over the full power range of the reactor. Some of the trip logic is only applicable at or near the full power operating limit. To account for this situation, some of the trip logic is overridden or "conditioned out" at low power levels. At high power levels the logic is "conditioned in" to the reactor trip calculations. The SRS contains tabular specifications for the power conditioning functions of several different plant parameters, each having its own different conditioning in and conditioning out values that appear in its function table as constants.

Below we provide a sample SRS function table for pressure conditioning logic.

$f\_PressCond(f\_EstPower : real, f\_PressCond_{-1} : bool) : bool =$

$f\_EstPower \leq k\_PressOUT$	$FALSE$
$k\_PressOUT < f\_EstPower < k\_PressIN$	$f\_PressCond_{-1}$
$k\_PressIN \leq f\_EstPower$	$TRUE$

The behavior of the function is illustrated in Figure 6. To eliminate "chatter" a deadband is used to create a hysteresis effect. When the estimated power $f\_EstPower$ drops below $k\_PressOUT$, the logic associated with the pressure sensor is "conditioned out" by setting $f\_PressCond$ to $FALSE$. When the power exceeds $k\_PressIN$, the logic is "conditioned in" and is used to evaluate the system. While $f\_EstPower$ is between $k\_PressOUT$ and $k\_PressIN$, the value of $f\_PressCond$ is left unchanged by setting it to its previous value, indicated by the "−1" subscript on the function name in the table and "No Change" in

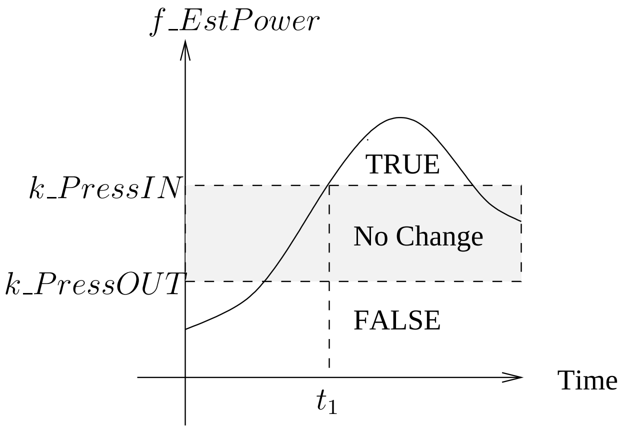

Figure 6. The *f_EstPower* deadband for power conditioning function *f_PressCond*.

Figure 6. For example, in the graph of $f\_EstPower$ in Figure 6, $f\_PressCond$ would start out $FALSE$, then become $TRUE$ at time t_1 and remain $TRUE$.

Upon considering the fact that there are several virtually identical SRS power conditioning functions that only differ in the names and values of their constants, the developer decided to reuse logic in the design specified by the SDD by writing one general power routine and passing in sensor parameters for different sensors. The following design was proposed by the developer for a general Power Conditioning function called PwrCnd:

$PwrCnd(Prev : bool, Power, Kin, Kout : posreal) : bool =$ (12)

$Power \le Kout$	$Kout < Power < Kin$	$Power \ge Kin$
$FALSE$	$Prev$	$TRUE$

In the above, $Prev$ is the argument for the power conditioning status of the particular sensor from the previous pass.

The PVS specification of the proposed general power conditioning function is shown in Figure 7. Note the similarity to the original table in (12).

```
PwrCnd(Prev:bool, Power, Kin, Kout:posreal):bool = TABLE
  %--------------------------------------------------%
  |[Power<=Kout | Power>Kout & Power<Kin | Power>=Kin]|
  %--------------------------------------------------%
  |   FALSE     |         Prev           |   TRUE    ||
  %--------------------------------------------------%
ENDTABLE
```

Figure 7. PVS Specification of general *PwrCnd* function

Typechecking the definition generates an unprovable disjointness TCC. The PwrCnd_TCC1 disjointness TCC and the unprovable sequent that results from trying to prove PwrCnd_TCC1 are shown in Figure 8. The first three formulas of

```
% Disjointness TCC generated (at line 19, column 54) for
 % TABLE
 % |[ Power <= Kout | Power > Kout & Power < Kin | Power >= Kin ]|
 % |    FALSE       |      Prev                  | TRUE         ||
 % ENDTABLE
% unfinished
PwrCnd_TCC1: OBLIGATION
 (FORALL (Kin: posreal, Kout: posreal, Power: posreal):
   NOT (Power <= Kout AND Power > Kout & Power < Kin)
       AND NOT (Power <= Kout AND Power >= Kin)
         AND NOT ((Power > Kout & Power < Kin) AND Power >= Kin));

PwrCnd_TCC1 :

[-1]    Kin!1 > 0
[-2]    Kout!1 > 0
[-3]    Power!1 > 0
[-4]    Power!1 <= Kout!1
[-5]    (Kin!1 <= Power!1)
  |-------
[1]     FALSE

Rule?
```

Figure 8. Disjointness TCC and resulting unprovable sequent for *PwrCnd*

the sequent contain the type information for Kin, $Kout$ and $Power$. All are of type $posreal = \{x : real | x > 0\}$. The names ending in "!1" are skolem constants - arbitrary constants of the appropriate type that are used to eliminate quantifiers from the formulas.

Following the procedure outlined in Section 2.2.2, we write down the characteristic formula for the unprovable sequent in Figure 8.

$$0 < Kin \wedge 0 < Kout \wedge 0 < Power \wedge Kin \leq Power \wedge Power \leq Kout \Rightarrow FALSE$$

With a slight abuse of notation, the above can be simplified to

$$\neg(0 < Kin \leq Power \leq Kout)$$

By taking $Kout = Power = Kin$, we obtain a counter example that makes the above characteristic formula false. One can easily verify that the counter example satisfies the conditions of two or more columns of the table for PwrCnd, thereby proving the table as defined does not properly specify a function. Substituting the counter example values into the tabular specification for $PwrCnd$, we see that the conditions for both the first and third result columns are satisfied producing the inconsistent result that $PwrCnd$ must be both $TRUE$ and $FALSE$ in this case.

The implicit (and undocumented) assumption that the developer made was that, as in the case of the SRS function in Figure 6, the conditioning in threshold exceeds the conditioning out threshold (i.e., $Kin > Kout$.). This led the designers to omit the counter example cases of the form $Kin \leq Kout$ from the table. Such an undocumented assumption has obvious potential danger in a setting where logic (and code) may be reused by other developers or maintenance staff who are unaware of the assumption.

The assumption can be made explicit either by including an assertion in the function definition that the SDV Tool then parses and translates into a proof obligation when the function is used, or by redesigning the function table to properly handle the $Kin \leq Kout$ case possibly by generating an error message.

Another method of making the assumption explicit is through the use of dependent typing to create a new version of the PwrCnd table that makes the assumed relation between Kin and Kout explicit as shown in Figure 9. The Disjointness and

```
PwrCnd(Prev:bool, Power, Kin:posreal, Kout:{x:posreal|x<Kin}):bool
 = TABLE
   %-----------------------------------------------%
   |[Power<=Kout|Power>Kout & Power<Kin|Power>=Kin]|
   %-----------------------------------------------%
   |   FALSE    |         Prev          |   TRUE   ||
   %-----------------------------------------------%
   ENDTABLE
```

Figure 9. Use of dependent typing to make $Kin > Kout$ assumption explicit.

Completeness TCCs for the table are proved automatically by PVS. This version of the PwrCnd function causes PVS to automatically generate a TCC requiring a proof that the $Kin > Kout$ relationship will not be violated whenever the function is used.

The above example illustrates how the use of tool supported tabular methods can be used to detect undocumented assumptions in a design. We note that the SESM Tool suite includes SRS and SDD development tools that can be used to perform checks similar to the PVS Disjointness and Completeness TCCs, directly on some of the simpler tables in the SRS and SDD documents. The requirements and design developers can use these tools as they create the documents to catch such problems before PVS is applied at the verification stage.

4.2. Abstraction Functions Effects and Tolerances

In this section we study the verification of a simplified pressure sensor trip that monitors a pressure sensor and is "tripped" when the sensor value exceeds a normal operating setpoint. As was the case with the power conditioning example above, the SRS specification of the pressure sensor trip also makes use of deadbands to eliminate chatter. The proposed SRS and SDD implementations for the sensor trip are give in Figure 10 by $f\_PressTrip$ and $PTRIP$, respectively. In the function definitions, f_PressTripS1 and PREV play corresponding roles as the arguments for the previous value of the state variable computed by the function.

Figure 10 also contains the supporting type, constant and abstraction function definitions for the verification block. The abstraction function $posreal2AI$ models the A/D conversion of the sensor values by taking the integer part of its input using the built in function $floor(x)$ from the PVS prelude file. It is used to map the real valued SRS input $Pressure$ to the discrete SDD input $PRES$ which has type AI. AI consists of the subrange of integers between 0 and 5000, denoted by $subrange(0, 5000)$ in Figure 10.

At the bottom of the specification, the theorem $Sentrip1$ is an example of a *block comparison theorem* that is used to prove a specific instance of the general block verification equation (11) that relates the SRS and SDD inputs and outputs. If $Pressure$ and $PRES$ were both real numbers, related by the identity map, then the block comparison theorem $Sentrip1$ would be easily proved, but in this case, where $PRES$ is a discrete input, attempting the block comparison produces the following unprovable sequent:

```
{-1}    real_pred(Pressure!1)
{-2}    Tripped?(f_PressTripS1!1)
{-3}    Pressure!1 < 2450
{-4}    floor(Pressure!1) <= 2400
  |-------
{1}    Pressure!1 <= 2400
```

Using sequent transformation (4)(i), we negate formula {1} and move it to the top half of the sequent and then apply transformation (1) to obtain the characteristic formula:

$$f\_PressTripS1 = Tripped \wedge$$
$$Pressure < 2450 \wedge floor(Pressure) \leq 2400 \wedge \neg(Pressure \leq 2400) \Rightarrow \bot$$

which simplifies to

$$\neg(f\_PressTripS1 = Tripped \wedge 2400 < Pressure < 2450$$
$$\wedge floor(Pressure) \leq 2400)$$

For any value of $Pressure$ in the open interval $(2400, 2401)$ when $f\_PressTrip$ was tripped in the previous pass, the above formula is $FALSE$. The problem

sentrip : THEORY
BEGIN

$\text{k\_PressSP} : \text{int} = 2450$
$\text{k\_DeadBand} : \text{int} = 50$

$\text{KDB} : \text{int} = \text{k\_DeadBand}$
$\text{KPSP} : \text{int} = \text{k\_PressSP}$

$\text{Trip} : \text{TYPE} = \{\text{Tripped}, \text{NotTripped}\}$
$\text{AI} : \text{TYPE} = \text{subrange}(0, 5000)$

$\text{f\_PressTrip}((\text{Pressure} : \text{posreal}), (\text{f\_PressTripS1} : \text{Trip})) : \text{Trip} =$ TABLE

$\text{Pressure} \leq \text{k\_PressSP} - \text{k\_DeadBand}$	NotTripped
$\text{k\_PressSP} - \text{k\_DeadBand} < \text{Pressure} \wedge \text{Pressure} < \text{k\_PressSP}$	f_PressTripS1
$\text{Pressure} \geq \text{k\_PressSP}$	Tripped

ENDTABLE

$\text{PTRIP}((\text{PRES} : \text{AI}), (\text{PREV} : \text{bool})) : \text{bool} =$ TABLE

$\text{PRES} \leq \text{KPSP} - \text{KDB}$	FALSE
$\text{KPSP} - \text{KDB} < \text{PRES} \wedge \text{PRES} < \text{KPSP}$	PREV
$\text{PRES} \geq \text{KPSP}$	TRUE

ENDTABLE

$\text{Trip2bool}((\text{TripVal} : \text{Trip})) : \text{bool} =$ TABLE

TripVal = Tripped	TRUE
TripVal = NotTripped	FALSE

ENDTABLE

$\text{posreal2AI}((x : \text{posreal})) : \text{AI} =$ TABLE

$x \leq 0$	0
$0 < x \wedge x < 5000$	$\text{floor}(x)$
$x \geq 5000$	5000

ENDTABLE

Sentrip1 : THEOREM
$(\forall\ (\text{Pressure} : \text{posreal}, \text{f\_PressTripS1} : \text{Trip}) :$
$\text{Trip2bool}(\text{f\_PressTrip}(\text{Pressure}, \text{f\_PressTripS1})) =$
$\text{PTRIP}(\text{posreal2AI}(\text{Pressure}), \text{Trip2bool}(\text{f\_PressTripS1})))$

END sentrip

Figure 10. Formatted PVS specification for pressure sensor trip example

occurs because whenever $2400 < Pressure < 2401$, the abstraction function $posreal2AI$ maps $Pressure$ to the same value as 2400, but when $f\_PressTripS1 = Tripped$, the SRS function $f\_PressTrip$ maps $Pressure$ values greater than 2400 to $Tripped$ while 2400 gets mapped to $NotTripped$. Therefore $\ker(posreal2AI) \not\leq \ker(f\_PressTrip)$, so by Claim 1 we know that there is no SDD design that can satisfy the block comparison theorem $Sentrip1$.

This is an example of when mathematical functional equality may be more strict than practically necessary. Due to the accuracy of the sensors, all input values have a tolerance of ± 5 units. In this case, the SDD function $PTRIP$ actually has acceptable behavior. Although the SESM tools do not yet support it, the functional 4-variable model they currently use can be easily extended to incorporate tolerances. In this case the input tolerances can be taken into account in PVS using existential quantification over a dependent type (see [19] for full details). Currently tolerances are taken into account using rigorous manual arguments.

4.3. Domain Coverage and Timing Limitations

The following example shows how the tools complement testing by covering all input cases using quantifier reasoning. It also demonstrates the current limitation of tool support for the verification of timing properties. The example deals with a trip status indicator that is used to flag when pressure sensor trip has occurred. Once every 5 seconds the Trip Computer transmits the status indicator flag. The transmitted indicator value depends upon the history of the pressure sensor trip in the previous 5 seconds. If there was a sensor trip at any time during the last 5 seconds, the transmitted indicator value is $TRUE$, otherwise, it is $FALSE$.

The original SRS specification of the trip status indicator is:

$f\_PressStatus(f\_PressTrip : Trip, f\_PressStatus_{-1} : bool, t_{now} : posreal) : bool =$

<table>
<tr><td colspan="2">$f\_PressTrip = Tripped$</td><td>$TRUE$</td></tr>
<tr><td rowspan="2">$NOT[f\_PressTrip = Tripped]$</td><td>$t_{now} MOD k\_Comdelay = 0$</td><td>$FALSE$</td></tr>
<tr><td>$t_{now} MOD k\_Comdelay \neq 0$</td><td>$f\_PressStatus_{-1}$</td></tr>
</table>

The interpretation of the above table is that if there is a sensor trip then the status indicator $f\_PressStatus$ is set to $TRUE$. When there is not a sensor trip, if it is time to transmit ($t_{now} MOD k\_Comdelay = 0$ corresponds to the case when the current time is a multiple of 5 seconds) then $f\_PressStatus$ is "cleared" by setting it to $FALSE$. Otherwise it is left at its previous value $f\_PressStatus_{-1}$.

To simplify the verification, we replace the timing condition by the boolean variable Transmit which is $TRUE$ when $t_{now} MOD k\_Comdelay = 0$. The formatted PVS for this version of the SRS function is shown in Figure 11. In addition to the tabular function implementations, the SDD contains the main program

$\text{f\_PressStatus}((\text{f\_PressTrip} : \text{Trip}), (\text{f\_PressStatusS1}, \text{Transmit} : \text{bool})) : \text{bool} \;=\; \textsc{table}$

f_PressTrip = Tripped	TRUE
$\neg$(f_PressTrip = Tripped) $\wedge$ Transmit	FALSE
$\neg$(f_PressTrip = Tripped) $\wedge$ $\neg$Transmit	f_PressStatusS1

ENDTABLE

$\text{STATUS}((\text{PRES} : \text{AI}), (\text{PREV} : \text{bool})) : \text{bool} \;=\; \textsc{table}$

PRES $\leq$ KPSP $-$ KDB	PREV
KPSP $-$ KDB $<$ PRES $\wedge$ PRES $<$ KPSP	PREV
PRES $\geq$ KPSP	TRUE

ENDTABLE

Status1 : THEOREM
($\forall$ (Pressure : posreal, f_PressTripS1 : Trip, f_PressStatusS1 : bool, Transmit : bool) :
f_PressStatus(f_PressTrip(Pressure, f_PressTripS1), f_PressStatusS1, Transmit) =
IF $\neg$(Transmit) THEN STATUS(posreal2AI(Pressure), f_PressStatusS1)
ELSE FALSE
ENDIF)

Figure 11. Formatted PVS input for the trip status indicator block comparison

thread that provides the function call sequence for the main program loop. The SDD status indicator logic is composed of two parts. The first part, determined by the function $STATUS$ as part of the pressure sensor trip module, mimics the sensor trip logic of $PTRIP$. The second part is performed in the main program thread by a conditional statement that checks a timer value to determine when it is time to transmit and reset the indicator value. This part of the status indicator computation is modeled by the IF-THEN-ELSE statement that is part of the block comparison theorem.

The definitions from Figure 11 can be appended to the specification in Figure 10. To avoid the abstraction function problems of Section 4.2, AI is changed to posreal and the abstraction function $posreal2AI$ is changed to the identity function. Attempting to prove the block comparison theorem $Status1$ results in several unprovable sequents, including the one below:

```
{-1}    real_pred(Pressure!1)
{-2}    Transmit!1
{-3}    Tripped?(f_PressTripS1!1)
  |-------
{1}    Pressure!1 <= 2400
```

The characteristic equation for the sequent simplifies to:

$$\neg(Transmit \wedge f\_PressTripS1 = Tripped \wedge Pressure > 2400) \qquad (13)$$

The counter example resulting from negating (13) corresponds to the case when there is a transmission and hence the SDD program thread clears the status indicator. On the other hand, the SRS status indicator remains $TRUE$ due to either (i) a current sensor trip directly forced by a high pressure value ($Pressure \geq 2450$) or (ii) the current pressure value remaining in the deadband ($2400 < Pressure < 2450$) when the pressure sensor trip was previously tripped. While the first case is not as serious since it will be corrected on the first pass after the transmission if $Pressure \geq 2450$, the second case is more serious since as long as $2400 < Pressure < 2450$, the SDD status indicator will be $FALSE$ while $PTRIP$ is $TRUE$! Both of these cases would requires the tester to use the specific test input that would be unlikely to occur in the reactor under normal operating conditions. While thorough unit testing should uncover this problem, detecting the problem during SDV allows it to be corrected prior to coding.

In this example we abstracted the timing properties to perform the verification and found a particular input sequence that the developers had not considered where the SRS and SDD differed. Timing properties are multi-pass properties that need to take into consideration scheduling and possibly sequences of previous inputs and states. Currently the SDV procedure requires separate manual rigorous arguments, though some preliminary work [20] has been done on adapting the timing verification techniques in [5] to handle these problems.

5. Related Work

This section provides a comparison with previous works focusing on application of tool supported formal methods to industrial control software problems. The discussion attempts to illustrate the distinguishing features of this work as well as point out its current limitations relative to these previous efforts.

The general SDV procedure and use of tabular methods at OPGI has been previously documented in [23, 29, 34]. Here we have provided further details about how the 4-variable model of [30] is specialized to our discrete time setting and then decomposed to facilitate application of the model to full scale industrial examples as outlined in [24]. More significantly, we have outlined how the procedure the has been adapted to provide practical, semi-automated tool support to the formal methods of the SDV procedure using the SESM Tool suite integrated with PVS. The tool support is now an integrated part of the overall software engineering process. They have been applied as part of the critical path of completion of the Darlington SDS Redesign project. The examples from Section 4 are drawn from this experience and help to illustrate the benefits and limitations of the method.

In this paper we have focused on the procedure and tools necessary to formally verify whether a software design meets its requirements. While applications of tool supported formal methods to industrial examples have been previously described in [11, 3, 21, 7], these case studies typically focus on requirements analysis. Also, in these examples applying tool supported formal method typically involved some reverse engineering of previously developed requirements documents. As a result these methods were not part of the production software engineering process but instead were viewed as pilot projects. The SDV procedure described here did not involve reverse engineering as an addon to the project but rather was an integral part of the overall software development process and, as such, was on the critical path to completion of the project. We will now examine each of these previous applications in further detail.

The Requirements State Machine Language (RSML) was first introduced by Leveson *et al.* in [21] to model the requirements of the onboard aircraft traffic collision avoidance system (TCAS II). It is based upon a state-chart like graphical notation augmented with tabular representation of transition guard conditions via the methods AND/OR tables. Requirements in RSML can be automatically checked for consistency and completeness [7].

Building on the original use of tabular methods in the A-7 [12], Heitmeyer *et al.* have made extensive use of the Software Cost Reduction (SCR) tabular methods supported by the "light-weight" SCR* tool suite [8, 9]. It has been used extensively for the creation and analysis of requirements for industrial and military software applications (e.g., [11]). In [11] the authors also describe work that has been done to allow users to to incorporate more heavy duty analysis tools such as the explicit state model checker SPIN [13] with SCR*.

Crow and Di Vito have used PVS' support for tabular specification and other functionality to formalize parts of the space shuttle's software system in [3]. Similar to the work described here, they employ a simple conventional abstract state machine semantics. The studies focused on requirements analysis while the translations into PVS were done manually by experienced PVS users. As a result, more manual effort was required to keep the PVS versions of the specification of the subsystems up to date with the main requirements documents.

The examples cited above produced formal methods specifications that were not (at least initially in the case of [21]) part of the main project documentation. Therefore keeping these formal documents up to date with subsequent revisions of the main project documents can become problematic (e.g., in [3] the authors note "convergence [of the 'official' documents and PVS code] was slow" due to "frequent and extensive" changes as the requirements were reviewed). The SDV procedure presented here results in simpler configuration control of the system documents. Only the word processor documents that are used by everyone involved in the project need to be modified when the software is revised. Once the input documents are prepared, the generation of the PVS is effectively a pushbutton operation. The problem of keeping the tool input up to date with the latest "official" version of the documents is avoided by having the SESM tools generate the theorem prover input directly from the documents. Since these are standard word processor documents, the document authors and maintainers do not need to learn any specialized formal methods tools, though they do have to adhere to a more rigid document format.

The completeness and consistency checking provided by the SESM tools and PVS is local in the sense of SCR as opposed to the global sense of RSML [7]. While SCR* implements similar table checks, our approach appears to be novel for its redundancy. The SESM tools provide a first check of the completeness and consistency of each table and then typechecking in PVS repeats the check, helping to reduce the potential for a single point of failure preventing the detection of any errors.

Considering the restricted application setting of modeling a single SDS digital controller with appropriately conditioned input signals, allows us to simplify the semantics of our underlying model. At the requirements level our underlying model is based upon a discrete time model where all inputs are "sampled" and then all state variables and outputs are simultaneously updated. In contrast, both the SCR and RSML semantics involve external events triggering sequences of specific internal transitions that are assumed to occur before the next event. While there are some cases where the state chart like notation RSML and more complicated semantics of both SCR and RSML would be better suited to specify the systems requirements, the vast majority of the SDS system requirements are easily modeled by tabular specifications with our simplified semantics. These are typically "single pass" properties such as the power conditioning example of Section 4.1 that rely on at most the value of current input and previous state.

The method presented here does not currently support the verification of "multi-pass", concurrent and real-time properties that would be more easily modeled with a formalism such as RSML or the timed automata employed in [2]. The SDV procedure currently handles these properties using manual verification. In the future, the SESM tool suite could try to support the verification of such properties by combining PVS' support for model checking tabular relations [25] with recent work on the use of PVS to verify real-time properties [2, 5, 20]. Ideally the SESM tools should be enhanced to allow the system designers to use alternative formal notations and tools where they are more appropriate.

The current combination of the SESM tools and PVS is lacking some of the more user friendly features of other tabular specification systems. In particular, the simulation, automatic counter example generation and dependency graph generation capabilities of SCR* would be very useful to developers and verifiers alike. Extending the SESM tools to interface with tabular methods tools such as SCR* could provide a cost effective way of adding these capabilities to the current verification system. If the Timed Automata Modeling Environment (TAME) is integrated with the SCR* as proposed in [2], this approach may also make it possible to hide some of the complexity of the PVS proof system behind a specialized user interface.

Currently the SESM Tool suite only supports functional verification. This is a severe limitation since often the SRS and SDD behaviors are not functionally equivalent but they are within specified tolerances as was the case in the sensor trip example of Section 4.2. In [19] we show how the functional 4-variable model of the SDV can be extended to a relational 8-variable model that provides for input and output tolerances on functional specifications. We also show how adding existential quantifiers over dependent types to the original block comparison theorems, allows PVS to easily handle variables with tolerances. These tolerances are already included in the SRS and SDD documents, hence future revisions of the SESM tools could parse these tolerances and incorporate them into the block comparison theorems.

In other related work, the authors of [18] use tool supported Colored Petri Nets (CPN) and PVS for requirements analysis of a reactor shutdown system. While CPN can be used to model multi-pass properties, the method makes extensive use of additional new axioms to model each CPN, thereby weakening PVS' guarantees of consistent extension of the logic [32]. We note that the tabular methods used in this work and [25] do not introduce any additional axioms.

6. Conclusions

Experience has shown that review and testing alone are not usually sufficient to guarantee the correct operation of a safety critical software system. This problem is partly due to the overwhelming amount of detail associated with a complete system. The SDV procedure of Section 3 provides a rigorous framework for the effective application of tabular methods to industrial software verification problems. The utility of the method is in part due to the use of simple algebraic properties to decompose the verification problem into manageable pieces in a way that limits the amount of manual effort required by the verifiers. Tool support also helps, making it easier to handle large volumes of material associated with such problems.

On the SDS Redesign project all team members used tabular notation as the main basis for specifying, designing, and verifying the safety critical software. The use of standard word processors to create the main project documents containing the tables reduces the learning curve required to implement a rigorous software engineering process and creates highly readable project documents. The SESM Tool suite aids in the document creation process by helping the document creators to debug tabular specifications. The tools extract the tabular specifications and generate PVS code to perform the block comparisons of the SDV procedure.

The use of PVS reduces human error in the mathematical proofs (e.g. dropping a negation) by taking care of much of the details of routine logical manipulations. Although only basic knowledge of sequent calculus and PVS is needed to obtain useful counter examples by interpreting the unprovable sequents that result when block comparisons fail, the SDV procedure has been designed so that only the verifiers need to understand PVS.

A nice feature of PVS is that proofs can be run at various levels of detail depending on their intended use. An initial verification pass can be automatically run with minimal output using a default high level proof strategy such as `(GRIND)` to detect problem areas. On the other hand, the final verification procedure proof output can be performed using only low level commands such as `(EXPAND ...)`, `(FLATTEN)`, `(SPLIT)` and `(LIFT-IF)` that can be more easily followed by a reviewer.

In conclusion, tabular methods have been successfully applied at OPGI to the development and verification of SDS Trip Computer Software. Carefully designed formal software development and verification procedures were central to this success by enabling the effective application of tool supported tabular methods as an integrated part of the complete software engineering process. While the initial results are encouraging, further work needs to be done in both academia and industry to address the current procedural and tool limitations regarding tolerances and timing and to provide a friendlier user interface.

Acknowledgements

The authors would like to thank Mike Viola of Ontario Power Generation and Jeff McDougall of JKM Software Technologies Inc. for their comments and constructive criticism regarding early drafts of this document. The authors would also like to thank the anonymous referees for their insightful and instructive criticisms of the manuscript.

References

1. Archer, M. and C. Heitmeyer: 1996, 'TAME: A Specialized Specification and Verification System for Timed Automata'. In: A. Bestavros (ed.): *Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96).* Washington, DC, pp. 3–6. The WIP Proceedings is available at `http://www.cs.bu.edu/pub/ieee-rts/rtss96/wip/proceedings`.
2. Archer, M., C. Heitmeyer, and S. Sims: 1998, 'TAME: A PVS Interface to Simplify Proofs for Automata Models'. In: *User Interfaces for Theorem Provers*. Eindhoven, The Netherlands. Informal proceedings available at `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.
3. Crow, J. and B. L. Di Vito: 1998, 'Formalizing Space Shuttle Software Requirements: Four Case Studies'. *ACM Transactions on Software Engineering and Methodology* **7**(3), 296–332.
4. Dutertre, B. and V. Stavridou: 1997a, 'Formal Requirements Analysis of an Avionics Control System'. *IEEE Transactions on Software Engineering* **23**(5), 267–278.
5. Dutertre, B. and V. Stavridou: 1997b, 'Requirements Analysis of Real-Time Control Systems Using PVS'. In: C. M. Holloway and K. J. Hayhurst (eds.): *LFM' 97: Fourth NASA Langley Formal Methods Workshop.* Hampton, VA, pp. 65–74. Available at `http://atb-www.larc.nasa.gov/Lfm97/proceedings/`.

6. Easterbrook, S., R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton: 1998, 'Experiences Using Lightweight Formal Methods for Requirements Modeling'. *IEEE Transactions on Software Engineering* **24**(1), 4–14. Special Section on Formal Methods in Software Practice.
7. Heimdahl, M. P. E. and N. G. Leveson: 1996, 'Completeness and Consistency in Hierarchical State-Based Requirements'. *IEEE Transactions on Software Engineering* **22**(6), 363–377. Special Section: Best Papers of the 17th International Conference on Software Engineering (ICSE-17).
8. Heitmeyer, C., A. Bull, C. Gasarch, and B. Labaw: 1995, 'SCR*: A Toolset for Specifying and Analyzing Requirements'. In: *Compass '95: 10th Annual Conference on Computer Assurance*. Gaithersburg, Maryland, pp. 109–122.
9. Heitmeyer, C., J. Kirby, and B. Labaw: 1997, 'The SCR Method for Formally Specifying, Verifying, and Validating Software Requirements: Tool Support'. In: *Proceedings of the 19th International Conference on Software Engineering*. pp. 610–611.
10. Heitmeyer, C., J. Kirby, B. Labaw, and R. Bharadwaj: 1998a, 'SCR*: A Toolset for Specifying and Analyzing Software Requirements'. In: *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, Vol. 1427 of *Lecture Notes in Computer Science*. pp. 526–531.
11. Heitmeyer, C., J. Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj: 1998b, 'Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications'. *IEEE Transactions on Software Engineering* **24**(11), 927–948.
12. Heninger, K. L.: 1980, 'Specifying Software Requirements for Complex Systems: New Techniques and Their Applications'. *IEEE Transactions on Software Engineering* **SE-6**(1), 2–13.
13. Holzmann, G. J.: 1997, 'The Model Checker SPIN'. *IEEE Transactions on Software Engineering* **23**(5), 279–295. Special Issue: Formal Methods in Software Practice.
14. Janicki, R. and R. Khédri: 2001, 'On a Formal Semantics of Tabular Expressions'. *Science of Computer Programming* **39**(2-3), 189–213.
15. Janicki, R., D. L. Parnas, and J. Zucker: 1997, 'Tabular Representations in Relational Documents'. In: C. Brink, W. Kahl, and G. Schmidt (eds.): *Relational Methods in Computer Science*, Advances in Computing Science. Springer Wien NewYork, Chapt. 12, pp. 184–196.
16. Jankowski, E. and J. McDougall: 1995, 'Procedure for the Specification of Software Requirements for Safety Critical Software'. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1001-PROC Rev. 1.
17. Joannou *et al.*, P.: 1995, 'Standard for Software Engineering of Safety Critical Software'. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1.
18. Koo, S., H. Son, and P. Seong: 1999, 'Mathematical Verification of a Nuclear Power Plant Protection System Function with Combined CPN and PVS'. *Journal of the Korean Nuclear Society* **31**(2), 157–171.
19. Lawford, M., J. McDougall, P. Froebel, and G. Moum: 2000, 'Practical application of functional and relational methods for the specification and verification of safety critical software'. In: T. Rus (ed.): *Proceedings Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000*, Vol. 1816 of *LNCS*. pp. 73–88.
20. Lawford, M. and H. Wu: 2000, 'Verification of real-time control software using PVS'. In: P. Ramadge and S. Verdu (eds.): *Proceedings of the 2000 Conference on Information Sciences and Systems*, Vol. 2. Princeton, NJ, pp. TP1–13–TP1–17.

21. Leveson, N. G., M. P. E. Heimdahl, H. Hildreth, and J. D. Reese: 1994, 'Requirements Specification for Process-Control Systems'. *IEEE Transactions on Software Engineering* **20**(9), 684–707.
22. McDougall, J. and J. Lee: 1995, 'Procedure for the Software Design Description for Safety Critical Software'. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1002-PROC Rev. 1.
23. McDougall, J., M. Viola, and G. Moum: 1994, 'Tabular Representation of Mathematical Functions for the Specification and Verification of Safety Critical Software'. In: *SAFECOMP'94: The 13th International Conference on Computer Safety, Reliability and Security*. Anaheim, California, pp. 21–30.
24. Moum, G.: 1997, 'Procedure for the Systematic Design Verification of Safety Critical Software'. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1003-PROC Rev. 1.
25. Owre, S., J. Rushby, and N. Shankar: 1997, 'Integration in PVS: Tables, Types, and Model Checking'. In: E. Brinksma (ed.): *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Vol. 1217 of *Lecture Notes in Computer Science*. Enschede, The Netherlands, pp. 366–383.
26. Parnas, D.: 1977, 'The use of precise specifications in the development of software'. In: *IFIP Congress*. pp. 861–867.
27. Parnas, D.: 1992, 'Tabular Representation of Relations'. Technical Report 260, Communications Research Laboratory, McMaster University.
28. Parnas, D.: 1995, 'Using Mathematical Models in the Inspection of Critical Software'. In: M. G. Hinchey and J. P. Bowen (eds.): *Applications of Formal Methods*, International Series in Computer Science. Prentice Hall, Chapt. 2, pp. 17–31.
29. Parnas, D. L., G. J. K. Asmis, and J. Madey: 1991, 'Assessment of safety-critical software in nuclear power plants'. *Nuclear Safety* **32**(2), 189–198.
30. Parnas, D. L. and J. Madey: 1995, 'Functional documents for computer systems'. *Science of Computer Programming* **25**(1), 41–61.
31. Rueß, H., N. Shankar, and M. K. Srivas: 1999, 'Modular Verification of SRT Division'. *Formal Methods in Systems Design* **14**(1), 45–73.
32. Rushby, J., S. Owre, and N. Shankar: 1998, 'Subtypes for Specifications: Predicate Subtyping in PVS'. *IEEE Transactions on Software Engineering* **24**(9), 709–720.
33. Shankar, N., S. Owre, and J. M. Rushby: 1993, 'PVS Tutorial'. Computer Science Laboratory, SRI International, Menlo Park, CA. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
34. Viola, M.: 1995, 'Ontario Hydro's Experience with New Methods for Engineering Safety Critical Software'. In: *SAFECOMP'95: The 14th International Conference on Computer Safety, Reliability and Security*. Belgirate, Italy, pp. 283–298.
35. Wassyng, A. and *et al.*: 1990, 'Choosing A Methodology For Developing System Requirements'. Ontario Hydro/AECL SD-2 Study Report.