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@opg.com)
P. Moum (g.moum@opg.com)
Ontario Power Generation, 700 University Ave., Toronto, ON, CANADA M5G 1X6

Abstract. This paper describes the use of tabular methods at Ontario Power Generation Inc. (OPG) 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 methods. Finally, the paper discusses related work, draws conclusions regarding the effectiveness of the methods and examines how its limitations can be addressed by further theoretical and applied work.

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

1. Introduction

This paper describes the application of tool supported tabular methods to the specification and verification of safety critical software. It provides a case study of the use of formal methods as an integral part of the software development process. The formal methods were applied as one part of a complete software engineering process that also includes hazards analysis, review (requirements, design and code) and testing (unit, integration, validation and reliability qualification). Some

* 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.

of these development steps were formal, while others were informal. Although the tabular notation facilitated all of these activities, we will concentrate on the role of tabular methods in the specification and functional verification of safety critical computer software.

In particular, the paper emphasizes how to make the formal aspects of the methods practical. The authors are of the opinion that it is important to make formal methods an intuitive and easily performed part of software development in order to improve the quality of both safety critical software and formal methods. This requires not only creating tools to support formal methods, but also making formal methods theoreticians aware of the implementation issues that tools and methods must address before they will be 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 for the Darlington Nuclear Generating Station Shutdown System (SDS) Trip Computer Software Redesign Project, hereafter referred to as the SDS Redesign Project.

Ontario Power Generation Inc. (OPG) and Atomic Energy of Canada Limited (AECL) have jointly defined a detailed engineering standard to govern the specification, design and verification of such safety critical software systems. The CANDU Computer Systems Engineering Centre of Excellence Standard for Software Engineering of Safety Critical Software [20] 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.*

To address this principle, the current implementation of the software engineering process makes extensive use of tool supported tabular methods. The process results in the production of a coherent set of documents that allow for limited 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 [31, 14, 33], 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., [26, 8, 12, 4]), 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 methods.

The SDS Redesign Project represents one of the first times that a production industrial 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 the particular formal methods for the Redesign Project were: (i) learning curve and ease of understanding, (ii) ability to provide tool support, and (iii) 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. Also, tables provide a mathematically precise notation with a formal semantics [17]. Other methods such as VDM or Z utilize unfamiliar notation and special languages with a significant learning curve [42]. 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 [7, 11]) 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 [39].

OPG had strong evidence that a verification procedure employing tabular methods would meet the requirements of the Redesign Project.
Prior to the Redesign Project, OPG 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 [28]. The creation of the specialized tools that allowed verification to be done with the help of PVS played a large role in making the methods 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 [36]. Further description of the tools support and rationale for OPG’s choice of tools is provided in Section 3.3.

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:

– Independent checks which are unaffected by the verifier’s expectations,

– Domain coverage through the use of tools that 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 [35] 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 methods 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 methods and examines how its limitations can be addressed by further theoretical and applied work.

2. Preliminaries

In this section we review the key principles of PVS goals and tabular specifications that will allow the reader to interpret the examples of Section 4.

2.1. PVS Proof Goals

PVS makes use of a Gentzen style sequent calculus to perform proofs. In this subsection we provide a brief description of a sequent or “proof goal” that is required to interpret the examples of Section 4. The reader is referred to [39] 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, i.e., provability in the logic’s proof system. Henceforth we will use \( \neg P_1, P_1 \land P_2 \) and \( Q_1 \lor Q_2 \) to denote negation, conjunction and disjunction respectively. We will use \( P_1 \Rightarrow Q_1 \) as an abbreviation for \( \neg P_1 \lor Q_1 \) and to reduce the number of parentheses required to write our formulas we assume the following decreasing order of precedence of operations: \( \neg, \land, \lor, \Rightarrow \).

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, \( P_1, P_2, \ldots, P_n \vdash Q_1 \lor Q_2 \lor \ldots \lor Q_m \), or equivalently \( P_1 \land P_2 \land \ldots \land P_n \vdash Q_1 \lor Q_2 \lor \ldots \lor Q_m \).
In sequent calculus this is written as:

\[
\begin{array}{c}
P_1 \\
\vdots \\
P_n \\
\hline
Q_1 \\
\vdots \\
Q_m
\end{array}
\]
or equivalently

\[
P_1 \land P_2 \land \ldots \land P_n \Rightarrow Q_1 \lor Q_2 \lor \ldots Q_m \tag{1}
\]

since in the “sequent” there are implicit \(\land\)’s between the premises and implicit \(\lor\)’s between the conclusions. The formula on the right of (1) is a logical theorem if the sequent is provable, thus we will refer to it as the characteristic formula of the sequent. For the special case when there are no premises, proving the sequent corresponds to showing that the disjunction of the conclusions is a logical theorem \((\vdash Q_1 \lor \ldots \lor Q_m)\), while proving the sequent when there are no conclusions, corresponds to proving that the premises are inconsistent \((P_1 \land \ldots \land P_n \vdash \bot)\).

2.2. PVS Support for 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 [30].

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

\[
f(x_1, \ldots, x_m) = \begin{array}{c|c|c|c|c|c|c}
  c_1 & c_2 & \ldots & c_n \\
  e_1 & e_2 & \ldots & e_n \\
\end{array}
\]
or

\[
\begin{array}{c|c|c|c|c|c|c}
  c_1 & c_2 & \ldots & c_n \\
  e_1 & e_2 & \ldots & e_n \\
\end{array}
\]

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: (i) Disjointness, which requires that the columns (rows) do not overlap. i.e., \(i \neq j \Rightarrow \neg(c_i \land c_j)\), and (ii) Completeness, which requires that at least one column (row) is applicable to every input. i.e., \((c_1 \lor c_2 \lor \ldots \lor c_n)\) 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 sign function and its equivalent tabular representation are:

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

For the purposes of this paper we will restrict ourselves to simple horizontal and vertical tables or minor variations thereof. More compact tabular representations such as Structured Decision Tables [28], Normal, Inverted and Vector Function tables [32] exist for the specification of complex functions. The fundamental notions of disjointness and completeness are easily generalized to these other types of 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 [37]. 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 proofs 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 various forms of a TABLE construct as a method of specifying function tables. The following PVS statements uses the horizontal condition table
version of the PVS TABLE construct to define the \textit{sign} function of Example 1. The \& below represents conjunction.

\[
\text{sign}(x:\text{real}): \{ i: \text{int} \mid i \geq -1 \& i \leq 1 \} = \text{TABLE} \\
\lfloor \begin{array}{ccc}
\mid & x<0 & x=0 & x>0 \\
\mid & -1 & 0 & 1 \\
\end{array} \rfloor \\
\text{ENDTABLE}
\]

Use of \texttt{TABLE} causes PVS to automatically generate Disjointness and Completeness TCCs (proof obligations). These can often be automatically discharged (proved) by PVS’ built in proof strategies. This is the case for the following TCCs generated by the above \texttt{TABLE} statement.

\%
\text{Disjointness TCC generated}
\texttt{sign\_TCC1: OBLIGATION FORALL (x: real):} \\
\quad \text{NOT (x < 0 AND x = 0) AND NOT (x < 0 AND x > 0)} \\
\quad \text{AND NOT (x = 0 AND x > 0);} \\
\%
\text{Coverage TCC generated}
\texttt{sign\_TCC2: OBLIGATION FORALL (x: real): x < 0} \\
\quad \text{OR x = 0 OR x > 0;}

Tables involving more complicated conditions may cause PVS’ built in procedures to fail to automatically discharge the resulting TCCs. In these cases the user can attempt to interactively discharge the proof obligations. As we will see in Section 4, when the built-in proof strategies and user intervention fail, the resulting unprovable sequent(s) can often provide useful information regarding the incompleteness or inconsistency of specifications.

\section{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 SDS 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. In particular, we concentrate on the verification of functional properties utilizing tabular notation. The reader is referred to [43] for details of the software process and notation, [24] for the decomposition of the proof obligations and [29] 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* [20] that was jointly developed by OPG 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 [43]. The software development stages relevant to this paper are governed by the Software Requirements Specification Procedure and the Software Design Description Procedure. 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 [18, 32] to specify the software’s behavior.

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 [20] 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 [35] 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:

\[REQ = OUT \circ SOF \circ IN\]  

is illustrated in the commutative diagram of Figure 1. Here \(REQ\) represents the SRS state transition function mapping the monitored variables statespace \(M\) to the controlled variables statespace \(C\). The function \(SOF\) represents the SDD state transition function mapping the behavior of the implementation input variables represented by statespace \(I\) to the behavior of the software output variables represented by the statespace \(O\). The mapping \(IN\) relates the specification’s monitored variables to the implementation’s input variables while \(OUT\) relates the implementation’s output variables to the specification’s controlled variables.
The underlying models of both the SRS and SDD are based upon Finite State Machines (FSM). For a large number of the system requirements, the FSMS are deterministic, allowing us to consider the special case when $REQ$ 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 1 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 [19, 27, 29, 24, 43].

3.2. Decomposing the Proof Obligations

In Figure 2 we decompose the proof obligation (3) to isolate the verification of hardware interfaces. The $M_p$ and $C_p$ state spaces are the software’s internal representation of the monitored and controlled variables, referred to as the pseudo-monitored and pseudo-controlled variables, respectively [24]. The proof obligations associated with SDV then become

\[
Abst_C \circ REQ = SOF_{req} \circ Abst_M \tag{4}
\]

\[
Abst_M = SOF_{in} \circ IN \tag{5}
\]

\[
id_C = OUT \circ SOF_{out} \circ Abst_C . \tag{6}
\]
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.

The controlled variable abstraction function is defined as \( \text{Abst}_C : C \rightarrow C_p \) which is seemingly the “wrong” direction. The proof obligation (6) forces \( \text{Abst}_C \) to be injective, preventing the possibility of trivial designs for \( \text{SOF}_{req} \) being used to satisfy the main obligation (4). The SDV procedure provides recourse for the case when there is not a 1-1 correspondence between \( C \) and \( C_p \) through the use of a pseudo-SRS that can be defined to more closely match the SDD [28, 43].

Typically the verification of a subsystem as represented by (4), the inner part of the commutative diagram, can be decomposed “horizontally” 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 3 where each equality of the form:

\[
\text{SOF}_i \circ \text{Abst}_{V_{i-1}} = \text{Abst}_{V_i} \circ \text{REQ}_i
\]

becomes a verification block. Here \( V_i \) and \( V_{ip} \) 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).

\[
\text{REQ}
\]

\[
\begin{align*}
\text{M} & \xrightarrow{\text{REQ}_1} V_1 \xrightarrow{\text{REQ}_2} V_2 \quad \ldots \quad V_{n-1} \xrightarrow{\text{REQ}_{n-1}} C \\
\text{M}_p & \xrightarrow{\text{SOF}_1} V_{1p} \xrightarrow{\text{SOF}_2} V_{2p} \quad \ldots \quad V_{(n-1)p} \xrightarrow{\text{SOF}_{n-1}} C_p \\
\end{align*}
\]

\( \text{SOF}_{req} \)

Figure 3. Horizontal (sequential) decomposition of proof obligations

The price paid for this vertical and horizontal decomposition is that for each block the verifier must provide a cross reference between the
internal variables and define the abstraction functions, \( \text{Abst}_{V_{i-1}} \) and \( \text{Abst}_{V_{i}} \). Now the benefits of defining all the abstraction functions, including \( \text{Abst}_C \), from top to bottom (SRS to SDD) in Figures 2 and 3 becomes more apparent. Defining all abstraction functions from top to bottom and then only performing the check for injectivity at the outputs embodied by (6) 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 [24].

3.3. Tool Support

An experience report of the first use of the above method prior to the use of support tools is detailed in [41]. 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, OPG 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., [7, 11]), combining light-weight tools with model-checkers and theorem provers can provide additional analysis and verification capabilities (e.g., [2, 12]). 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 common 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 4 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 create and debug the software requirements in the SRS which is then used by the developers to create the software design in the SDD. Finally, both the SRS and SDD are used by the verifiers to create the Design Verification Report (DVR). 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
Figure 4. Relationship between tools and documents of the SDV process

the verifier with guidance provided by the SRS and SDD documents. This information flow between documents is indicated by the dotted lines in Figure 4.

Next, the word processor is used to create Rich Text Format (RTF) versions of these three documents that become input for the SESM 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 PVS input files (b001.pvs, b002.pvs, ...), one for each verification block, containing the block’s function and type definitions together with the block comparison theorem corresponding to equation (7). PVS is then used to typecheck and prove the block comparison theorems in each file. The results of the PVS block comparison proofs are then integrated into the DVR and any discrepancies between the SRS and SDD are duly noted.

Typechecking of the tabular function definitions within PVS provides one of the checks of completeness and consistency of tabular specifications in our method. In addition, for many of the tabular functions definitions appearing in the SRS, SDD or DVR, the SESM tools provide simple completeness and consistency checks 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. The tabular checks implemented by the SESM tools consist of rudimentary propositional reasoning applied to the coverage and disjointness conditions for tables with up to 15 boolean conditions. The SESM tools perform these checks by first transforming the table into a structured decision table [28] with one column for each of the possible combinations of values of the boolean conditions in the table. The coverage check then verifies that in every case, either an action is specified, or the case is marked as impossible, in which case the tools rely upon the table author, and then PVS, to verify that the conjunction of conditions in the column are mutually
exclusive. The determinism check attempts to find logical combinations of conditions that are captured by more than one column, in which case it verifies that the columns have consistent actions. Additionally the tools check the documents to insure that all table inputs are defined and that there are no circular dependencies of functions in the requirements specified by the SRS. While the SESM tools currently lack even the most basic linear arithmetic decision procedures found in PVS and other theorem proving systems, the simple checks described above generate warning messages that help to flag potential problems for the verifier to scrutinize more closely with the full power of PVS.

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, 5, 25, 23] and functional properties involving tolerances [24]. Additional reasons for choosing PVS were its direct support for tabular methods integrated with theorem proving and model-checking [30] and extensive type-checking capabilities which can be used to detect software errors [37]. 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.

3.4. SDS Redesign Tool Usage

The tools and procedures described here have been applied successfully to the SDS Redesign Project, which was completed in early 1999. The project consisted of a complete redesign of 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 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 reports (DVR) were each over 600 pages once completed (excluding PVS input and output), although 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 input 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 each received a week-long training course 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. This process resulted in roughly 70% of the over 200 functional blocks from the two software designs of the Redesign Project being being formally verified using the SDV Tool together with PVS. The remainder of the verification blocks that did not involve straight forward block comparisons, requiring additional reasoning about the program’s main execution thread and timing constraints, were handled by rigorous manual arguments. Future versions of the tools may be extended to handle some of these remaining cases using techniques such as those outlined in [25, 23].

As the SESM tools are further refined and the verifiers gain more experience, this part of the SDV procedure will require less time. Also, with the ability to rerun proofs in batch mode, it is 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, OPG 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 would 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.

\[
\begin{array}{|c|c|}
\hline
\text{f\_PressCond}(\text{f\_EstPower : real, f\_PressCond}_{-1} : \text{bool}) : \text{bool} = \\
\text{\text{\text{\text{\text{\textbf{f\_PressCond}}}}}_{-1}} \\
\text{f\_EstPower < k\_PressOUT} & \text{FALSE} \\
k\_PressOUT < \text{f\_EstPower} < k\_PressIN & \text{f\_PressCond}_{-1} \\
k\_PressIN < \text{f\_EstPower} & \text{TRUE} \\
\hline
\end{array}
\]

The behavior of the function is illustrated in Figure 5. To eliminate “chatter”, a deadband is used to create a hysteresis effect. When the estimated power \(\text{f\_EstPower}\) drops below \(k\_\text{PressOUT}\), the logic associated with the pressure sensor is “conditioned out” by setting \(\text{f\_PressCond}\) to FALSE. When the power exceeds \(k\_\text{PressIN}\), the logic is “conditioned in” and is used to evaluate the system. While \(\text{f\_EstPower}\) is between \(k\_\text{PressOUT}\) and \(k\_\text{PressIN}\), the value of \(\text{f\_PressCond}\) 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 5. For example, in the graph of \(\text{f\_EstPower}\) in Figure 5, \(\text{f\_PressCond}\) 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 conditioning routine and passing in sensor parameters for different sensors. The following design was proposed by the developer for a general Power...
Conditioning function called \texttt{PwrCnd}:

\begin{verbatim}
PwrCnd(Prev : bool, Power, Kin, Kout : posreal) : bool = Power ≤ Kout | Kout < Power < Kin | Power ≥ Kin
\end{verbatim}

\begin{tabular}{|c|c|c|c|c|}
\hline
FALSE & Prev & TRUE \\
\hline
\end{tabular}

\begin{equation}
\text{(8)}
\end{equation}

In the above, \texttt{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 6. Note the similarity to the original table

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

\textit{Figure 6. PVS Specification of general PwrCnd function}

in (8).

Typechecking the definition generates an unprovable disjointness TCC. The \texttt{PwrCnd\_TCC1} disjointness TCC and the unprovable sequent that results from trying to prove \texttt{PwrCnd\_TCC1} are shown in Figure 7. The first three formulas of the sequent contain the type information for \texttt{Kin}, \texttt{Kout} and \texttt{Power}. All are of type \texttt{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.

Recalling the definition of the characteristic equation of a sequent in Section 2.1, we write down the characteristic formula for the unprovable sequent in Figure 7.

\[
0 < \text{Kin} \land 0 < \text{Kout} \land 0 < \text{Power} \land \text{Kin} \leq \text{Power} \land \text{Power} \leq \text{Kout} \Rightarrow \text{FALSE}
\]

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

\[
-(0 < \text{Kin} \leq \text{Power} \leq \text{Kout})
\]

By taking \texttt{Kout} = \texttt{Power} = \texttt{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 \texttt{PwrCnd}, thereby proving that the table as defined does not
% Disjointness TCC
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 7. Disjointness TCC and resulting unprovable sequent for PwrCnd

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 5, 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 ≤ 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 as-
sertion 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 ≤ 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 8. The Disjointness and Completeness TCCs for this

\[
PwrCnd(\text{Prev}: \text{bool}, \text{Power}, \text{Kin}: \text{posreal},
    \text{Kout}: \{ x: \text{posreal} | x < \text{Kin} \}): \text{bool} = \text{TABLE}
\]

\[
%-----------------------------------------------%
| \text{Prev} | \text{FALSE} | \text{TRUE} |
%-----------------------------------------------%
\]

Figure 8. Use of dependent typing to make \( \text{Kin} > \text{Kout} \) assumption explicit.

table are proved automatically by PVS. This version of the \( PwrCnd \)
function causes PVS to automatically generate a TCC whenever the
function is used requiring a proof that the \( \text{Kin} > \text{Kout} \) relationship will
not be violated.

The above example illustrates how the use of tool supported tab-
ular methods can detect undocumented assumptions in a design. As
previously noted in Section 3.3, 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 some 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 sen-
or 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 9 by \( f_{\text{PressTrip}} \) and \( \text{PTRIP} \), respectively. In the function
definitions, \( f_{\text{PressTripS1}} \) and \( \text{PREV} \) play corresponding roles as the
arguments for the previous value of the state variable computed by the
function.

Figure 9 also contains the supporting type, constant and abstraction
function definitions for the verification block. The abstraction function
\( \text{posreal2AI} \) models the A/D conversion of the sensor values by taking
the integer part of its input using the built in function \( \text{floor}(x) \) from the
PVS prelude file. It is used to map the real valued SRS input \( \text{Pressure} \)
sentrip: THEORY
BEGIN

k_PressSP : int = 2450
k_DeadBand : int = 50

KDB : int = k_DeadBand
KPSP : int = k_PressSP

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

f_PressTrip((Pressure : posreal), (f_PressTripS1 : Trip)) : Trip = TABLE
| Pressure ≤ k_PressSP − k_DeadBand | NotTripped |
| k_PressSP − k_DeadBand < Pressure ∧ Pressure < k_PressSP | f_PressTripS1 |
| Pressure ≥ k_PressSP | Tripped |
ENDTABLE

PTRIP((PRES : AI), (PREV : bool)) : bool = TABLE
| PRES ≤ KPSP − KDB | FALSE |
| KPSP − KDB < PRES ∧ PRES < KPSP | PREV |
| PRES ≥ KPSP | TRUE |
ENDTABLE

Trip2bool((TripVal : Trip)) : bool = TABLE
| TripVal = Tripped | TRUE |
| TripVal = NotTripped | FALSE |
ENDTABLE

posreal2AI((x : posreal)) : AI = TABLE
| x ≤ 0 | 0 |
| 0 < x ∧ x < 5000 | floor(x) |
| x ≥ 5000 | 5000 |
ENDTABLE

Sentrip1: THEOREM
∀ (Pressure : posreal, f_PressTripS1 : Trip):
Trip2bool(f_PressTrip(Pressure, f_PressTripS1)) =
PTRIP(posreal2AI(Pressure), Trip2bool(f_PressTripS1))
END sentrip

Figure 9. Formatted PVS specification for pressure sensor trip example
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 9.

At the bottom of the specification, the theorem $\text{Sentrip1}$ is an example of a block comparison theorem that is used to prove a specific instance of the general block verification equation (7) 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 $\text{Sentrip1}$ would be easily proved, but in this case, where PRES is a discrete input, attempting the block comparison produces the following unprovable sequent:

\[
\begin{align*}
-1 & \quad \text{real\_pred(Pressure!1)} \\
-2 & \quad \text{Tripped?(f\_PressTripS1!1)} \\
-3 & \quad \text{Pressure!1 < 2450} \\
-4 & \quad \text{floor(Pressure!1) <= 2400} \\
\hline
1 & \quad \text{Pressure!1 <= 2400}
\end{align*}
\]

The characteristic formula of this sequent is:

\[
f_{\text{PressTripS1}} = \text{Tripped} \land \\
\text{Pressure < 2450} \land \text{floor(Pressure) <= 2400} \Rightarrow \text{Pressure <= 2400}
\]

negating and simplifying we obtain:

\[
\neg(f_{\text{PressTripS1}} = \text{Tripped} \land 2400 < \text{Pressure} < 2450 \\
\land \text{floor(Pressure) <= 2400})
\]

For any value of Pressure in the open interval (2400, 2401) when $f_{\text{PressTrip}}$ was tripped in the previous pass, the above formula is FALSE. The problem occurs because whenever $2400 < \text{Pressure} < 2401$, the abstraction function $\text{posreal2AI}$ maps Pressure to the same value as 2400, but when $f_{\text{PressTripS1}} = \text{Tripped}$, the SRS function $f_{\text{PressTrip}}$ maps Pressure values greater than 2400 to Tripped while 2400 gets mapped to NotTripped.

The above result indicates not only that the current version of PTRIP does not behave as specified by $f_{\text{PressTrip}}$, but, in fact, it is not possible to find any definition of PTRIP to satisfy the requirements. In order to factor $\text{Trip2bool} \circ f_{\text{PressTrip}}$ through the equivalence kernel of $\text{posreal2AI} \times \text{Trip2bool}$, this second function must retain at least as much information about its domain as $\text{Trip2bool} \circ f_{\text{PressTrip}}$. More precisely, if we are to meet the requirements then for all positive real numbers $P_1$ and $P_2$ and Trip values $t_1$ and $t_2$, if $\text{posreal2AI} \times$
Trip2bool\((P_1, t_1)\) = posreal2AI \times Trip2bool\((P_1, t_1)\) then it must be the case that Trip2bool \circ f_{PressTrip}(P_1, t_1) = Trip2bool \circ f_{PressTrip}(P_2, t_2). From the preceding discussion we know that this property fails to hold is for \(P_1 = 2400, P_2 = 2400.5\) and \(t_1 = t_2 = Tripped\). Thus it appears that the developer in charge of implementing the \(f_{PressTrip}\) requirement was assigned an impossible task.

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 \(\text{PTRIP}\) 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 [24] 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 a 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 \textsc{true}, otherwise, it is \textsc{false}.

The original SRS specification of the trip status indicator is:

\[
\begin{align*}
\text{f}_{\text{PressStatus}}(\text{f}_{\text{PressTrip}} : \text{Trip}, \text{f}_{\text{PressStatus}}_{-1} : \text{bool}, \text{t}_{\text{now}} : \text{posreal}) : \text{bool} = \\
\begin{array}{|c|c|}
\hline
\text{f}_{\text{PressTrip}} = \text{Tripped} & \text{TRUE} \\
\hline
\text{f}_{\text{PressTrip}} \neq \text{Tripped} & \\
\begin{array}{c}
\text{t}_{\text{now}} \text{MODk}_{\text{Comdelay}} = 0 \\
\text{t}_{\text{now}} \text{MODk}_{\text{Comdelay}} \neq 0
\end{array} & \text{FALSE} \\
\hline
\end{array}
\end{align*}
\]

The interpretation of the above table is that if there is a sensor trip then the status indicator \(f_{\text{PressStatus}}\) is set to \textsc{true}. When there is not a sensor trip, if it is time to transmit \((t_{\text{now}} \text{MODk}_{\text{Comdelay}} = 0\) corresponds to the case when the current time is a multiple of 5 seconds\) then \(f_{\text{PressStatus}}\) is “cleared” by setting it to \textsc{false}. Otherwise it is left at its previous value \(f_{\text{PressStatus}}_{-1}\).
To simplify the verification, we replace the timing condition by the boolean variable Transmit which is TRUE when $t_{now} \mod k_{\text{Comdelay}} = 0$. The formatted PVS for this version of the SRS function is shown in Figure 10. In addition to the tabular function implementations, the

$$f_{\text{PressStatus}}(f_{\text{PressTrip}} : \text{Trip}, f_{\text{PressStatusS1}} : \text{bool}) : \text{bool} = \begin{array}{|c|c|}
\hline
f_{\text{PressTrip}} = \text{Tripped} & \text{TRUE} \\
\hline
\neg (f_{\text{PressTrip}} = \text{Tripped}) \land \text{Transmit} & \text{FALSE} \\
\neg (f_{\text{PressTrip}} = \text{Tripped}) \land \neg \text{Transmit} & f_{\text{PressStatusS1}} \\
\hline
\end{array}$$

ENDTABLE

$$\text{STATUS}(\langle \text{PRES} : \text{AI} \rangle, \langle \text{PREV} : \text{bool} \rangle) : \text{bool} = \begin{array}{|c|c|}
\hline
\text{PRES} \leq \text{KPSP} - \text{KDB} & \text{PREV} \\
\text{KPSP} - \text{KDB} < \text{PRES} \land \text{PRES} < \text{KPSP} & \text{PREV} \\
\text{PRES} \geq \text{KPSP} & \text{TRUE} \\
\hline
\end{array}$$

ENDTABLE

$\text{Status1} : \ \text{THEOREM}$

$\forall (\text{Pressure} : \text{posreal}, f_{\text{PressTripS1}} : \text{Trip}, f_{\text{PressStatusS1}} : \text{bool}, \text{Transmit} : \text{bool}) :$

$f_{\text{PressStatus}}(f_{\text{PressTrip}}(\text{Pressure}, f_{\text{PressTripS1}}), f_{\text{PressStatusS1}}, \text{Transmit}) =$

$\begin{cases} 
\text{IF} \neg (\text{Transmit}) \text{ THEN STATUS(posreal2AI(\text{Pressure}), f_{\text{PressStatusS1}}) } \\
\text{ELSE FALSE} \\
\text{ENDIF}
\end{cases}$

$\text{Figure 10. Formatted PVS input for the trip status indicator block comparison}$

SDD contains the main program 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 $\text{STATUS}$ as part of the pressure sensor trip module, provides the basic sensor trip logic. It is not an exact copy of the $f_{\text{PressStatus}}$ table since the transmit timing is handled by the calling code in the main program thread - the second part of the SDD status indicator logic. The main program thread evaluates a conditional statement that checks a timer value to determine when it is time to transmit and then 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 10 can be appended to the specification in Figure 9. To avoid the abstraction function problems of Section 4.2, $\text{AI}$ is changed to $\text{posreal}$ and the abstraction function $\text{posreal2AI}$ is changed to the identity function. Attempting to prove the block
comparison theorem Status1 results in several unprovable sequents, including the one below:

\begin{verbatim}
{-1} real_pred(Pressure!1)
{-2} Transmit!1
{-3} Tripped?(f_PressTripS1!1)

|-------

{1} Pressure!1 <= 2400
\end{verbatim}

The characteristic equation for the sequent simplifies to:

\[ \neg (Transmit \land f_{PressTripS1} = Tripped \land Pressure > 2400) \] (9)

The counter-example resulting from negating (9) 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 \cite{25, 23} has been done on adapting the timing verification techniques in \cite{6} 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 OPG has been previously documented in [28, 34, 41, 43]. Here we reviewed details of [24] on how the 4-variable model of [35] is specialized to our functional, discrete time setting and then decomposed to facilitate application of the model to full scale industrial examples. A similar hardware hiding decomposition of the 4-variable model has been independently developed in [40] motivated by software product lines for embedded control systems. In this work we have outlined how the procedure has been adapted to provide practical, semi-automated tool support to the formal methods of the SDV procedure using the SESM Tool suite together with PVS.

Software practitioners have clearly indicated the need to automate routine tasks in order to effectively and reliably develop software [16]. The application of formal methods similarly needs to become a largely automated process with tools of even better quality than those used for building and testing software. Knight et al. hypothesize that by incorporating formal methods tools into existing software packages such as off the shelf office suites and other software engineering tools, formal methods might be able to overcome their lack of “superstructure” and become more widely used in industry [21]. To support the claim, they create a toolset with preliminary support for combined analysis of formal and natural language elements at the specification level and apply it to a maritime software standards document. The toolset uses Z as the formal notation, Framemaker as the WYSIWYG environment for document production and Z/Eves automated reasoning system [38] for analysis. The work described here was performed in a production setting at both the requirements validation and design verification stages. It confirms the hypothesis that the integration of formal methods with existing desktop applications greatly increases the utility of the formal methods.

We have also 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 [12, 4, 26, 8], these case studies typically focus on requirements analysis. Also, in these other examples, applying tool supported formal methods 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 any such reverse engineering.
Rather tool support was planned from the start of the project and the software engineering process was adapted to facilitate tool usage. As an integral part of the overall software development process, the tool supported formal methods were on the critical path to completion of the Darlington SDS Redesign project. All three examples from Section 4 are drawn from this experience and help to illustrate the benefits and limitations of the methods. Below we provide a more detailed comparison with each of the previously referenced applications.

The Requirements State Machine Language (RSML) was first introduced by Leveson et al. in [26] 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 AND/OR tables. Requirements in RSML can be automatically checked for consistency and completeness [8].

Building on the original use of tabular methods in the A-7 [14], Heitmeyer et al. have made extensive use of the Software Cost Reduction (SCR) tabular methods supported by the “light-weight” SCR* tool suite [9, 10]. It has been used extensively for the creation and analysis of requirements for industrial and military software applications (e.g., [12]). In [12] the authors also describe work that has been done to allow users to incorporate more heavy duty analysis tools such as the explicit state model checker SPIN [15] 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 [4]. 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 [26]) 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 [4] 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 input files 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 are local checks that can be used to help infer the determinism and coverage of the system’s global state transition relation. When a table makes use of a term defined by another table, the completeness and consistency checks may use the term’s definition. This approach is similar to that employed by SCR [13]. There is no explicit reasoning about the overall correctness of the system based upon the system architecture as can be done with RSML [8]. 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 periodically “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. Because of the use of previous state values in the tables, it is possible to model SCR and RSML style semantics in our setting through the use of “stuttering” or “null” transitions. This can be viewed as a case of modeling an asynchronous transition systems by a synchronous transition systems as described in [3]. While there are certainly 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, even at the requirements level, rely on at most the value of current input and previous state.

The methods presented here do not currently support the verification of “multi-pass”, concurrent and real-time properties that would
typically involve inductive proofs or model-checking. These properties would be more easily modeled in formalisms 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 [30] with recent work on the use of PVS to verify real-time properties [2, 6, 25, 23]. 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. A special user interface could be developed to hide some of the complexity involved in using PVS to prove the block comparison theorems of the SDV procedure in much the same manner that the Timed Automata Modeling Environment (TAME) hides some of the complexity of using PVS to prove state invariants of automata models [2].

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 [24] 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 [22] 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 [37]. We note that the tabular methods used in this paper and [30] 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 tool supported tabular methods to industrial software verification problems. The utility of the methods 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 the 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 common 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 OPG 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 OPG, Jeff McDougall of Systemware Innovation Corp. and Wolfram Kahl of McMaster for their comments on drafts of this work. The anonymous referees feedback helped us to significantly improve upon the original manuscript.

References


