Formal Verification of Real-time Software

By

Hongyu Wu, B.A.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
McMaster University

MASTER OF SCIENCE(2000)	McMaster University
(Computer Science)	Hamilton, Ontario

TITLE:	Formal Verification of Real-time Software
AUTHOR:	Hongyu Wu, B. A. (Peking University)
SUPERVISORS:	Dr. Mark Lawford and Dr. David L. Parnas
NUMBER OF PAGES:	x, 129

Abstract

The verification of functionality of the input/output logic properties often composes the majority of software requirements analysis [12]. Automated theorem provers (ATPs) such as SRI's Prototype Verification System (PVS) have been successfully used in the formal verification of functional properties. However, such functional methods are not readily applicable to the verification of the real-time software requirements.

The thesis continues the research summarized in [13], focusing on extending functional verification methods to the verification of real-time control properties through the development of a PVS library for the specification and verification of real-time control system. More specifically, we extend the PVS "Clocks" and "Held_For" theories originally defined in [4] and [13], and refine the PVS Real-Time method (*PVS-RT*) outlined in [13]. New developments of the thesis include the definition of strong clock induction and several lemmas regarding real-time properties. These definitions, when combined with PVS's support for the tabular methods [15] of Parnas *et al.* [8, 18] provide a useful environment for the specification and verification of basic real-time control properties.

To illustrate the utility of the *PVS-RT* method, we perform the verification of two timing blocks of an industrial real-time control system. The PVS specification and proof techniques are described in sufficient details to show how errors or invalid assumptions are detected in the proposed implementation and the original specifications. Finally we prove that corrected versions of the implementation satisfy the updated versions of the specifications.

Acknowledgements

First of all, I am very grateful to my supervisor, Dr. Mark S. Lawford, who was always here for me. Discussions with him lead to many ideas described in the thesis. And he patiently read through many draft of this thesis. Without his guidance and support, this thesis would not have been possible.

I would also like to express my sincere thanks to Dr. David L. Parnas, my co-supervisor, for his kind supervision and support throughout my graduate studies.

I would like to express my appreciation to Dr. William M. Farmer and Dr. Jeffrey I. Zucker for agreeing to review my thesis and for their thoughtful comments.

Next, I thank all the friends I've made in McMaster University. The list is too long to mention but their friendship helped to make my stay in MAC a really happy one.

Special thanks to my wife, Ying, and my family for their love, encouragement and support.

Finally, I would like to acknowledge the financial support received from the Natural Sciences and Engineering Research Council (NSERC) and Bell Canada.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The verification of functionality of the input/output logic (where the output only depends upon the current inputs to the system) often composes the majority of the system requirement analysis [13]. Automated theorem provers (ATPs) such as SRI's Prototype Verification System (PVS) have been successfully used in the formal verification of these types of functional properties [6, 12, 24]. However, such methods are not immediately applicable to the verification of real-time software requirements of modern computer control systems [12, 13]. One of the major differences between the real-time requirements and the functional requirements is that the current output typically depends upon the past input history. In addition to requiring that the proper output is produced for a given input sequence, real-time software also requires the output to be produced at the correct time. Since they relate timed sequences of inputs to timed sequences of outputs, real-time requirements tend to be more difficult to design and verify. A formal (mathematically sound) method therefore would significantly aid the design and verification process.

1.1 Related Work

Researchers have proposed many innovative formal methods for verifying real-time system software.

In [19], Peters demonstrates a technique for generating a *monitor* from real-time system requirements documentation and illustrates how such a *monitor* can be used to observes the system behavior and determine if it is consistent with a given specification.

Archer and Heitmeyer [3] apply PVS to specify and reason about real-time systems modeled as timed automata (TAME). They use a continuous time setting to define data type *time* in PVS.

Dutertre and Stavridou of [4] show how PVS can be used to formalize the software requirements and to verify real-time properties in an avionics project. Rather than using a continuous time setting, [4] defines a discrete time "Clocks" theory. As we will see, this discrete time setting can be used to specify and verify the software for a digital control system that periodically samples its inputs and updates its state variables and outputs. While a more general continuous (dense) real-time semantics would allow for the modeling of interacting concurrent real-time systems, for our purposes the simpler discrete time setting of [4] is sufficient.

Based on the untimed functional verification method summarized in [12], Lawford and Wu [13] develop the PVS library for the specification and verification of real-time control systems. In particular, the PVS "Held_For" theory is built upon the "Clocks" theory of [4], where one is allowed to determine whether a timing condition has held for a fixed period of time. Most significantly, the authors of [13] propose a formal method called PVS Real-Time (*PVS-RT*). Compared to [3] and [4] where studies focused on requirements analysis only, the *PVS-RT* method can be used to formally model subsystems with hard real-time requirements (i.e., "timing blocks") and verify that the proposed implementation meet these requirements. As a result, errors or unexpected behavior are often discovered in the implementation and/or specification. Designers can then make modifications to address the errors and these modifications can then be formally verified prior to system testing. Thus the formal verification efforts provide the system engineers with a higher degree of confidence in the system's correctness.

In this thesis, the examples make use of Parnas' tabular representations of mathematical functions [8, 18] to specify the software's behavior. The tables provide a mathematically precise notation (see [9] for the formal semantics) for specifying software requirements and implementation details in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike. Previous work by [1] focused on formalizing tables to facilitate the construction and use of support tools such as McMaster's Table Tool System (TTS). In [25], Fu developed a tool to translate tables between the different formats used in TTS and those employed by Ontario Hydro [14]. In related work, Jing [10] describes an extension to TTS that allows the tool to export tabular specifications PVS so that the theorem prover could be used to check the tables for completeness and consistency conditions.

1.2 Motivation

The thesis continues the research summarized in [13], focusing on extending untimed functional method to the *PVS-RT* method. Moreover, we will try to "provide guidance on the use of theorem provers to specific classes of problems and in developing libraries to facilitate specification and verification", which is regarded as the main practical issue of the formal verification of real-time software in [4].

1.3 Scope

The thesis extends the PVS "Clocks" originally defined in [4]. A theorem for strong induction on clock values is provided so that one is allowed to use a broader induction hypothesis in theorem proofs. Additional developments also include stating and proving of several lemmas regarding timing properties that can reduce the efforts required in the proofs. These new definitions and developments, when combined with PVS's support for the tabular methods [15] of Parnas *et al.* [8, 18] provide a useful environment for the specification and verification of basic real-time control properties.

The thesis also illustrates the use of the *PVS-RT* method through the formal verification of two subsystems of an industrial real-time control system. The PVS specification and proof techniques are described in sufficient details to show how the errors or invalid assumptions are detected in the proposed implementation and the original specification. The specification and implementations are then amended and successfully verified.

1.4 Organization of the Thesis

Chapter 2 gives a brief introduction to tabular specification, sequent calculus and the PVS theorem prover. It also provides an overview of the Systematic Design Verification (SDV) procedure, which is the basis of the real-time software verification problems posed in Chapters 4 and 5. Chapter 3 provides the PVS specification of the clock theory and Held_For theory, including the definitions of clock operators, clock inductions (weak and strong) and Held_For operator. Some of the important lemmas "pulled out" from the actual theorem proofs are also included in this chapter. Chapters 4 and 5 describe the application of the *PVS-RT* method to two timing block comparisons, the channel trip sealed-in block and the channel trip reset block. Chapter 6 summarizes the the benefits and limitation of the current methods and proposes some future work.

Chapter 2

Notation and Preliminaries

The notation used throughout this thesis is defined in this chapter. We also review the key principles of PVS, sequent calculus and tabular specifications that will allow reader to interpret the examples. Finally we provide a brief overview of the Systematic Design Verification (SDV) procedure, which is the basis of the real-time software verification problem posed in the later chapters.

2.1 Notation

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 $\neg P_1$, $P_1 \wedge P_2$, $P_1 \vee P_2$, $P_1 \Rightarrow Q_1$ and $P_1 \Leftrightarrow Q_1$ to denote negation, conjunction, disjunction, implication and "if and only if" statement, respectively.

The special symbols $\top$ and $\bot$ will be used to denote boolean constants *true* and *false*.

The symbols $\forall$ and $\exists$ stand for the universal quantifier and the existential quantifier respectively.

To reduce the number of parentheses required to write our formulas, we assume the following decreasing order of precedence of operations, which is also the order of precedence in PVS:

$$\neg,\quad \wedge,\quad \vee,\quad \Rightarrow,\quad \Leftrightarrow,\quad \begin{matrix}\forall\\ \exists\end{matrix}$$

For a set V_i, we 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 use $g \circ f$ to denote *functional composition*, i.e., $g \circ f(v_1) = g(f(v_1))$.

The `typewriter font` is used to represent the names of PVS theorem , function or PVS command; the *italic font* in Chapter 4 and 5 is used to represent a variable.

2.2 PVS

The Prototype Verification System (PVS) is an environment for specification and verification that has been developed at SRI International's Computer Science Laboratory. The system consists of a specification language, a parser, a type checker, and an interactive theorem prover. The specification language is based on higher-order logic with a richly expressive type system so that a number of semantic errors in specification can be caught by the type checker. If the specification contains no additional axioms, typechecking can guarantee the conservative extension, i.e., the user defined theorems introduce no inconsistencies to the system [20].

The PVS theorem prover [22] consists of a powerful collection of inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. One can use the primitive inferences to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover.

2.3 Sequent Calculus

Sequent Calculus was introduced by Gentzen in 1935 as a way to present the syntax of logical systems. In this section we review the key principles of the PVS sequent calculus and the associated PVS commands. The examples and descriptions in the following section are largely based upon [12, 2].

2.3.1 Sequent

The basic structure of the PVS sequent calculus is a *sequent*, which can be defined as follows:

Definition 2.3.1 *A sequent in the sequent calculus is an ordered pair (* Γ, Δ*) of sets of formulas in higher order logic and is written* $\Gamma \vdash \Delta$*. Here* Γ *is called the antecedent,* Δ *is called the consequent, and* $\vdash$ *denotes syntactic entailment [21].*

There are two special cases:

(1) $\Gamma \vdash$ for the sequent $(\Gamma, \emptyset)$, meaning Γ can syntactically entail any set, i.e. Γ is *inconsistent*;

(2) $\vdash \Delta$ for the sequent $(\emptyset, \Delta)$, meaning Δ can be syntactically entailed by any set, i.e. Δ is a syntactic theorem.

Sequents in PVS are represented as a list of antecedents and consequents separated by a turnstile. There are implicit $\wedge$'s among the antecedents and implicit $\vee$'s among the consequents. Below is a typical sequent in PVS:

```
 [-1] P(x)
 {-2} Q(y)
|--------
 {1}  R(x)
 [2]  S(y)
```

Here P(x) and Q(y) are antecedents and R(x) and S(y) consequents. The numbers in square brackets, e.g., [-1], or in braces, e.g., {-2}, are used to name the corresponding formulas. A negative number represents the antecedent whereas a positive number denotes the consequent. The numbers in brackets [] indicate formulas that are unchanged in a subgoal from the parent goal whereas the numbers in braces {} serve to highlight those formulas which are either new or different from those of the parents.

This sequent can be translated into the following logical expression:

$$P(x) \wedge Q(y) \Rightarrow R(x) \vee S(y) \tag{2.1}$$

This logical expression is also called the *characteristic formula* [12]. Each *characteristic formula* can always be simplified to a *disjunctive normal form (DNF)*, i.e., as a disjunction of logical terms. For instance, the expression (2.1) can be simplified in the following way:

$$
\begin{aligned}
& P(x) \wedge Q(y) \Rightarrow R(x) \vee S(y) && && (2.2)\\
\Leftrightarrow\ & \neg(P(x) \wedge Q(y)) \vee R(x) \vee S(y) && \text{(Rule of the Conditional)} && (2.3)\\
\Leftrightarrow\ & \neg P(x) \vee \neg Q(y) \vee R(x) \vee S(y) && \text{(De Morgan's Rule)} && (2.4)
\end{aligned}
$$

The expression (2.4) is a *DNF* which is true when one of its disjuncts is true.

2.3.2 Proofs in sequent calculus

The ultimate goal of the PVS proof process is determining that the sequent under examination is true by determining that one of its disjuncts is true.

In the actual proof, a sequent can be discharged in any of the following cases:

(1) FALSE is an antecedent.
In this case, $\neg FALSE$ becomes one of the disjuncts. $\neg FALSE$ reduces to true making the entire disjunction true.

(2) TRUE is a consequent.
In this case, TRUE becomes one of the disjuncts making the entire disjunction true.

(3) Formula A is both an antecedent and a consequent.
In this case, both A and $\neg A$ are present in the disjunction. $A \vee \neg A$ is a tautology and reduces to true making the entire disjunction true.

When these three cases appear in the sequent, PVS will recognize them as trivially true and the proof will be finished showing "Q.E.D" at the end.

The entire objective of PVS is to manipulate sequents to obtain one of the afore mentioned cases. We will now outline some of the basic manipulation rules in the sequent calculus, together with the associated, low-level PVS commands.

- Eliminate the conjunction ($\wedge$) and the disjunction ($\vee$) in the antecedent and consequent respectively.

$$\begin{array}{|l} P_1 \wedge P_2 \\ \hline Q_1 \vee Q_2 \end{array} \quad \Longleftrightarrow \quad \begin{array}{|l} P_1 \\ P_2 \\ \hline Q_1 \\ Q_2 \end{array} \tag{2.5}$$

The PVS command associated with this rule is (*flatten*)

- Eliminate the conjunction($\wedge$) in the consequent and the disjunction ($\vee$) in the antecedent by splitting them into two subgoals.

$$\text{(i)} \quad \begin{tikzcd} & \begin{array}{|l} \Gamma \\ \hline Q_1 \wedge Q_2 \\ \Delta \end{array} \arrow[dl] \arrow[dr] & \\ \begin{array}{|l} \Gamma \\ \hline Q_1 \\ \Delta \end{array} & & \begin{array}{|l} \Gamma \\ \hline Q_2 \\ \Delta \end{array} \end{tikzcd} \qquad \text{(ii)} \quad \begin{tikzcd} & \begin{array}{|l} P_1 \vee P_2 \\ \Gamma \\ \hline \Delta \end{array} \arrow[dl] \arrow[dr] & \\ \begin{array}{|l} P_1 \\ \Gamma \\ \hline \Delta \end{array} & & \begin{array}{|l} P_2 \\ \Gamma \\ \hline \Delta \end{array} \end{tikzcd} \tag{2.6}$$

The PVS command associated with this rule is (*split*)

- Eliminate negation in the sequent.

$$\text{(i)} \begin{array}{|l} \Gamma \\ \hline \neg Q \\ \Delta \end{array} \quad \Longleftrightarrow \quad \begin{array}{|l} \Gamma \\ Q \\ \hline \Delta \end{array} \qquad \text{(ii)} \begin{array}{|l} \Gamma \\ \neg P \\ \hline \Delta \end{array} \quad \Longleftrightarrow \quad \begin{array}{|l} \Gamma \\ \hline P \\ \Delta \end{array} \tag{2.7}$$

Normally, PVS system will move the negation to the opposite part automatically when we begin the related theorem proof.

- Eliminate the implication ($\Rightarrow$) in the consequent.

$$\begin{array}{|l} \Gamma \\ \hline P_1 \Rightarrow P_2 \\ \Delta \end{array} \quad \Longleftrightarrow \quad \begin{array}{|l} \Gamma \\ P_1 \\ \hline P_2 \\ \Delta \end{array} \tag{2.8}$$

The PVS command associated with this rule is (*flatten*)

- Eliminate the implication ($\Rightarrow$) in the antecedent by splitting it into two subgoals.

$$\begin{tikzcd} & \begin{array}{|l} P_1 \Rightarrow P_2 \\ \Gamma \\ \hline \Delta \end{array} \arrow[dl] \arrow[dr] & \\ \begin{array}{|l} P_2 \\ \Gamma \\ \hline \Delta \end{array} & & \begin{array}{|l} \Gamma \\ \hline P_1 \\ \Delta \end{array} \end{tikzcd} \tag{2.9}$$

The PVS command associated with this rule is (*split*).

- Eliminate the universal quantifier in the consequent and the existential quantifier in the antecedent.

$$\text{(i)} \begin{array}{|l} \Gamma \\ \hline \forall x_1, \dots, x_n : Q \\ \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ \hline Q[c_1/x_1, \dots, c_n/x_n] \\ \Delta \end{array} \tag{2.10}$$

$$\text{(ii)} \begin{array}{|l} \Gamma \\ \exists x_1, \dots, x_n : Q \\ \hline \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ Q[c_1/x_1, \dots, c_n/x_n] \\ \hline \Delta \end{array} \tag{2.11}$$

The PVS command associated with this rule is (*skolem fnum (const)*), where *fnum* is a PVS formula number identifying the formula to be skolemized and *const* is a list of skolem constants ($c_1, \dots, c_n$). Note that a skolemized constant c_i must be the *fresh* constant, i.e. it must not already appear in the sequent.

- Eliminate universal quantifier in the antecedent and existential quantifier in the consequent.

$$\text{(i)} \begin{array}{|l} \Gamma \\ \forall x_1, \dots, x_n : Q \\ \hline \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ Q[t_1/x_1, \dots, t_n/x_n] \\ \hline \Delta \end{array}$$

$$\text{(ii)}\ \begin{array}{|l} \Gamma \\ \hline \exists x_1, \ldots, x_n : Q \\ \Delta \end{array} \Rightarrow \begin{array}{|l} \Gamma \\ \hline Q[t_1/x_1, \ldots, t_n/x_n] \\ \Delta \end{array}$$

A universally quantified variable in (i) or an existential quantified variable in (ii) can be instantiated by any term of the same type. The PVS command associated with this rule is (*inst fnum term*), where *fnum* is a formula number identifying the formula to be replaced and *term* is a list of instantiation constants $(t_1, \ldots, t_n)$. Note that the quantified formula is not copied while using (*inst fnum term*). That is, the quantified formula is hidden in the current sequent but it is still there. We can use the PVS command (*reveal fnum*) to see the hidden formula or use (*inst-cp fnum term*) to instantiate the quantifier and at the same time leave the original quantifier in the sequent.

2.3.3 Unprovable sequents and counter examples

Suppose we can make all disjuncts of the sequent (2.4) false by assigning:

$$\neg P(x) = \neg Q(y) = R(x) = S(y) = FALSE$$

then the whole sequent is false. In this case, the sequent is unprovable. One can easily verify that this assignment provides a counter example showing the original formula (2.1) is not a logical theorem.

2.4 Tables and PVS Support for Tables

This section briefly describes the tabular specification of functions, table formats and PVS support for the tabular expressions.

2.4.1 Tabular specification of functions

Multi-dimensional mathematical expression, called tables, have proven to be useful for specifying certain kinds of relations and functions [18].

Considering the function $sign(x)$, where x is an integer variable.

$$sign(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases} \tag{2.12}$$

Using tabular expression, this function can be specified as follows:

$$sign(x) = \begin{array}{|c|c|c|} \hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \tag{2.13}$$

Function (2.13) properly defines a *total* function (a function which is defined for all of its domain) because it satisfies the following two conditions:

Disjointness: requires the conjunction of a pairwise formulas to be false. In other words, columns(rows) do not overlap, i.e., $((x < 0) \wedge (x = 0)) \vee ((x = 0) \wedge (x > 0)) \vee ((x < 0) \wedge (x > 0))$ is always false.

Completeness: requires that disjunction of a set of formulas to be true, i.e., $(x < 0) \vee (x = 0) \vee (x > 0)$ is always true.

Tables provide a mathematically precise notation for the formal semantics in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike [12]. Tables make the cases explicit, allowing checks that none of th columns overlaps and that all the possibilities are considered [9].

One can weaken the disjointness conditions by requiring that where there is overlap between columns (rows), the columns (rows) produce the same results. For instance,

$$g(x) = \begin{array}{|c|c|} \hline x <= 0 & x >= 0 \\ \hline x & 2 * x \\ \hline \end{array} \tag{2.14}$$

the function (2.14) violates the *disjointness condition*, but it is still a total function because when the two columns overlap (i.e., $x = 0$), the expressions x and $2 * x$ return the same result, i.e., 0.

In practice, overlapping columns (rows) of tables may create confusion. To change a function at a value where columns (rows) overlap requires changing the table in two or more places instead of one. In addition, expressions which are mathematically equivalent at certain values may not be computational equivalent due to roundoff errors, etc. The same specification may results in different outcomes depending upon how the designer implements the specification. Therefore we will utilize the strict disjointness condition above when considering whether a table properly defines a function

2.4.2 Table formats

Several table formats have been found to be useful in the specification of relations and functions [18, 9, 14, 1, 25]. In this subsection, however, we will focus on the table formats known as *Labeled Complex Horizontal Condition Tables* and *Structured Decision Tables* [14], which are heavily used in the remaining chapters.

As mentioned earlier, a table is a multidimensional mathematical expression, consisting of *grid(s)* and *header(s)*. One can informally think a *grid* as a multidimensional array and a *header* as a one dimensional vector [25].

Simple Horizontal/Vertical Condition Table

The `sign` function (2.13) is a simple vertical condition table, whose table grid structure can be presented as follows:

H_1
H_2

In this table, both the *condition header* (H_1) and the *action header* (H_2) are one dimensional vectors. The `sign` function can also be rewritten by using a simple hor-

$$(1)\quad g(x) = \begin{array}{|c||c|}\hline x < 0 & -1 \\ \hline x = 0 & 0 \\ \hline x > 0 & 1 \\ \hline \end{array} \qquad\qquad (2)\quad \begin{array}{||c||c||}\hline H_1 & H_2 \\ \hline \end{array}$$

Figure 2.1: A simple horizontal condition table

izontal conditional table. Figure 2.1 shows its tabular expression and the associated table grid structure.

Labeled Complex Horizontal Condition Table

As its name shows, a *Labeled Complex Horizontal Condition Table (LCHCT)* is a kind of horizontal condition table. An example of this type of table is shown in Table 2.1.

Table 2.1: A labeled complex horizontal condition table

			Result
Condition			c_sealedin
init(t)			TRUE
NOT init(t)	(param_trip) Held for (seal_in_delay)		TRUE
	NOT [(param_trip) Held for (seal_in_delay)]	manual_reset	FALSE
		¬manual_reset	No Change
G_1			H_1

This table has one grid and one header, i.e., G_1 and H_1. Here G_1 represents the *condition grid* that may contain predicate expressions, condition macro names, and variable names. H_1 represents the *action header* whose entry contains condition macro terms, enumerated data and constants. Here *Condition* and *Result* are the labels for the table. Compared to the simple horizontal condition table, a LCHLT table has more complex cell structure in the *condition grid* where conjunction grids are involved. When the conjunction of atomic propositions in a given row of the *Condition* columns is $TRUE$, then *param_trip* is set to the *Result* value for that row.

Structured Decision Table

The example of a *structured decision table (SDT)* is shown in Table 2.2. As we will see, a *SDT* consists of two grids G_1, G_2 and two headers H_1, H_2.

H_1	G_1
H_2	G_2

H_1 represents the *condition header* that may contain predicate expressions, condition macro names, and variable names. G_1 represents the *condition grid* whose entries contain "T"s, "F"s, condition macro terms, and enumerated data. H_2 represents the *action header* that contains all the possible actions. G_2 represents the *action grid* that contains check mark "X"s, indicating that when the set of conditions in the same column of G_1 are met, the action statement in the same row of H_2 is the result action [25]. For instance, Column 8 can be read as:

$$\begin{array}{ll} & \neg(m\_ChanReset = e\_Stuck) HELD\_FOR(k\_Stuck) \\ \wedge & \neg(m\_ChanReset = e\_KeyOn) HELD\_FOR(k\_Debounce) \\ \wedge & (m\_ChanReset = e\_NotKeyOn) HELD\_FOR(k\_Debounce) \\ \wedge & f\_ChanReset\_s1 = woff \\ \Rightarrow & f\_ChanReset = e\_ClearTrip \end{array}$$

Note that a dash sign "-" in the cell denotes a "don't care" condition and an asterisk (*) placed next to the corresponding case number denotes an impossible case or *unreachable state*, e.g., Column 6 in Table 2.2.

Table 2.2: Structured decision table

H_1 G_1

Condition Statement	1	2*	3	4	5	6*	7	8	9	10	11	12	13	14
(m_ChanReset = e_KeyOn) HELD_FOR (k_Stuck)	T	F	F	F	F	F	F	F	F	F	F	F	F	F
(m_ChanReset = e_KeyOn) HELD_FOR (k_Debounce)	-	T	T	T	T	T	F	F	F	F	F	F	F	F
(m_ChanReset = e_NotKeyOn) HELD_FOR (k_Debounce)	-	T	F	F	F	F	T	T	T	T	F	F	F	F
f_ChanReset_s1	-	-	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr
Action Statements														
f_ChanReset = e_WaitForOn							X		X	X	X			X
f_ChanReset = e_WaitForOff			X	X								X		
f_ChanReset = e_ResetStuck	X				X								X	
f_ChanReset = e_ClearTrip								X						

H_2 G_2

2.4.3 PVS support for tables

The PVS specification language provides facilities for declaring types, functions, variables, constants and formulas. It can also represent the functions defined as tabular expressions.

To ensure that a function is well-defined, the tabular specification of the function causes PVS to generate completeness and disjointness proof obligations called *Type Correctness Conditions (TCCs).* Most of these proof obligations will be automatically discharged using PVS built-in proof strategies. The remaining complex TCCs require the user to input the proof rules interactively. In the case that the TCC cannot be proved either automatically or manually, the unprovable sequent can often provide useful information regarding the incompleteness or inconsistency of the specifications. 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.

PVS provides the COND and TABLE constructs to define tables.

PVS COND Construct

The PVS COND construct is a multi-way extension to the *if-then-else* construct of PVS. The following PVS statement uses the COND construct to define the previous $sign(x)$ function.

```
signs: TYPE = { i: int | i >= -1 & i <= 1}
  sign_cond(x:int): signs =
    COND
      x<0 -> -1,
      x=0 -> 0,
      x>0 -> 1
    ENDCOND
```

This generates the following TCCs, both of which are proved by PVS's built-in proof strategy.

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

Note that symbol % in PVS starts the line of the comments. In this case, *sign_cond_TCC3* is a disjointness TCC, responding to the *disjointness condition*; *sign_cond_TCC4* is a coverage TCC, responding to the *completeness condition*. Depending upon the specification, PVS may generate other TCCs regarding subtypes, recursive function termination, etc. The interested reader may see [20] for further detail.

PVS TABLE Construct

The PVS specification language provides various TABLE constructs to make the prover input more readable. For instance, here is another definition of the *sign*(x) function:

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

Note that the first delimiter after the keyword `TABLE` (case insensitive) must be |[, rather than the simple |, and the final delimiter on the first row is]| rather than ||. PVS translates the TABLE constructs into PVS COND constructs for typechecking and proving purposes.

PVS TABLE Construct with X Operator

It is possible to represent more complex tables such as a two-dimensional table or a *labeled complex horizontal condition table* through the use of COND or nested CONDs construct. The following is the PVS definition of Table (2.1) using COND:

```
c_sealed_in(t): RECURSIVE bool = COND
  init(t) -> true,
  NOT init(t) AND Held_For(param_trip,seal_in_delay)(t) -> true,
  NOT init(t) AND NOT Held_For(param_trip,seal_in_delay)(t)
    AND manual_reset(t) -> false,
  NOT init(t) AND NOT Held_For(param_trip,seal_in_delay)(t) AND
    NOT manual_reset(t) -> c_sealed_in(pre(t))
  ENDCOND
  MEASURE rank(t)
```

The PVS TABLE construct can also represent more complex tables. To simplify tabular expression in PVS, we use *X operator* proposed in [24], that takes a list of predicates as its arguments and return boolean variables. Below is the PVS definition of Table 2.1 using TABLE construct and *X operator*.

```
b:var bool
true(b):bool = b
false (b):bool = not b
DC (b):bool = true

c_sealed_in_table(t): RECURSIVE bool =
 LET X = (LAMBDA (a: pred[bool]),
          (b: pred[bool]),
          (c: pred[bool]):
        a(init(t)) &
        b(Held_For(param_trip,seal_in_delay)(t)) &
        c(manual_reset(t)))
 IN TABLE
  %    init(t)
  %      |    Held_For(param_trip,seal_in_delay)(t)
  %      |        |   manual_reset(t)
  %      |        |      |
  %      v        v      v      Channel trip sealed_in
    %----------|-------|--------|---------------------%
    | X(true ,     DC   ,  DC)   |     true           ||
    %----------|--------|-------|---------------------%
    | X(false ,  true   ,  DC)   |     true           ||
    %----------|--------|-------|----------------------%
    | X(false ,  false,   true) |     false          ||
    %----------|--------|-------|----------------------%
    | X(false,   false,   false)|  c_sealed_in(pre(t))||
    %----------|--------|-------|----------------------%
 ENDTABLE
 MEASURE rank(t)
```

In the above function, we first define boolean functions *true (b), false (b)* and *DC (b)* which will be used in the *X operator*. Note that $DC(b)$ denotes the "don't care" statement and always return *true* not matter what value of b is. A *LET* clause introduces the *X operator*, i.e., $X(a, b, c)$. Here a, b and c are predicates, i.e., *pred*[*bool*], whose domain is a boolean variable. The predicate values are assigned by the LAMBDA expression. For instance, $a = init(t)$. Therefore, $X(false, false, true)$ in the third row of above table can be interpreted as follows:

when

$$\neg init(t) \wedge \neg Held\_For(param\_trip, seal\_in\_delay)(t) \wedge manual\_reset(t),$$

the function `c_sealed_in_table(t)` returns *false*.

With the help of the X operator, the PVS TABLE construct can represent complex tables in a more visually attractive format than the COND construct. In the verification of the channel trip sealed-in block of this thesis, we present the SRS function in COND construct as well as in TABLE construct with *X operator*. Furthermore, we prove both expressions are equivalent. The interested reader is referred to Appendix E.2 for further details.

2.5 SDV Procedure Overview

In this section we review the Systematic Design Verification (SDV) procedure used in [12] that is the basis of the *PVS-RT* method posed in Chapters 4 and 5. The method makes use of a form of Parnas' tabular representations of mathematical functions [8, 18] to specify the software's behavior.

2.5.1 Overview

We assume the underlying models of both the Software Requirements Specification (SRS) and the Software Design Description (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. Thus the SRS provides a high level description of the required system behavior while the SDD provides the implementation details to implement the required behavior.

Software engineering standards for safety critical software, such as [11], require that the design be formally verified against the requirement and then the code be formally verified against the implementation to ensure that the implementation meets the requirements. For the purposes of this thesis we will concentrate on the process of verifying the design against the requirement only. This process is known as a Systematic Design Verification (SDV).

The objective of the SDV process 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. The process employed in [12] is based upon a variation of the *4 - variable model* of [17] 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:

$$\forall M[M \in domain(NAT) \Rightarrow (REQ(M) = OUT(SOF(IN(M))))] \tag{2.15}$$

is illustrated in the commutative diagram of Figure 2.2.

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

Figure 2.2: Commutative diagram for 4 variable model

Here **M** denotes the *monitor variables*. NAT represents the system constraints placed on the *monitor variables*. REQ represents the SRS state transition function mapping the *monitored variables* to the *controlled variables* and updated state represented by **C**. The function SOF represents the SDD state transition function mapping the behavior of the implementation *input variables* represented by state space **I** to the behavior of the software *output variables* represented by the state space **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*.

In the 4-variable model of [17], each of the 4 "variable" state spaces **M**, **I**, **O**, and **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 state space. 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 timed behavior of the variable m_i can be represented as a function $m_i^t : \mathbb{R} \rightarrow Type(m_i)$, where $m_i^t(x)$ is the value of the quantity m_i at time x. We can then take **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 and have it more closely model the dynamics of a digital control system that samples its inputs and updates its outputs at regular intervals, 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 sequence". 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 sequence 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 sequence 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 $K \in \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the positive reals. Thus observation z corresponds to time $(z * K)$. The value of m_i at any point between two observations (i.e., in the range of time $(z * K, (z+1) * K)$) is defined to be equal to $m_i(z)$.

The verification of real-time properties requires us to consider REQ and SOF as mapping from sequences of inputs to sequences of outputs.

2.5.2 Decomposing the proof obligations

In the verification of timing blocks, the proof obligation can be decomposed to isolate the verification of hardware interfaces [12]. Figure 2.3 shows the decomposition

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

Figure 2.3: Decomposition of proof obligations

schematic of the proof obligation of (2.15). The $\mathbf{M}_p$ and $\mathbf{C}_p$ state spaces are the software's internal representation of the monitored and controlled variables. The proof obligations associated with SDV then become

$$Abst_C \circ REQ = SOF_{req} \circ Abst_M \tag{2.16}$$

$$Abst_M = SOF_{in} \circ IN \tag{2.17}$$

$$id_{\mathbf{C}} = OUT \circ SOF_{out} \circ Abst_C \tag{2.18}$$

The equation (2.16) represents a comparison of the functionality of the system and should contain most of the complexity of the system. It is the main obligation. The equation (2.18) uses the *identity map* , i.e., $id_{\mathbf{C}}$, which is defined in Section 2.1. The equations(2.17) and (2.18) represent comparisons of the hardware hiding software of the system. These two obligations are often fairly straightforward and are discharged manually. The interested reader is referred to [12, 14] for further details.

Chapter 3

Real-time Modeling and Specification in PVS

The model of time employed by the *PVS-RT* method builds upon a discrete time "Clocks" theory originally defined by Dutertre and Stavridou in [4]. While the model of time used in [4] allows for multiple clocks of different frequency and continuous time functions, we restrict ourselves to discrete time functions of a single clock frequency.

This chapter describes the underlying real-time setting that will be used to model systems in PVS and then details the PVS implementations of the clock induction definition and the HELD_FOR timing function. The chapter concludes with some PVS lemmas regarding timing properties.

3.1 Clocks and Basic Clock Operators

We will consider time to be the set of non-negative real numbers. Then for a positive real number K, we define a clock of period K, denoted $clock_K$, to be a set of "sample instances"

$$clock_K := \{t_0, t_1, t_2, \dots, t_n, \dots\} = \{0, K, 2K, \dots, nK, \dots\}$$

To identify the initial clock value and thereby specify initial system states, we define the *init* predicate which is TRUE only at t_0:

$$init(t_n) := \begin{cases} TRUE, & n = 0 \\ FALSE, & \text{otherwise} \end{cases}$$

Identifying the initial clock value allows one to define recursive functions that use t_0 as the base case and then define the system state at any clock value in terms of the system state at the previous clock value. To formalize the notion of "previous clock value" and aid in proving termination properties of recursive functions defined over $clock_K$, we define the *rank* of t_n to be n. Formally: $rank_K : clock \to \mathbb{N}$ where $t_n \mapsto n$.

When defining recursive functions that have a clock of period K, for a particular instance of time (clock value) it is often convenient to be able to refer to the next sample time or previous sample time. To this purpose we can define $next_K$ and pre_K operators on the elements of $clock_K$ as follows:

$$\begin{aligned} pre_K(t_n) &:= \begin{cases} t_{n-1}, & n \geq 1 \\ \text{undefined}, & \text{otherwise} \end{cases} \\ next_K(t_n) &:= t_{n+1} \end{aligned}$$

When the value of K is unambiguous from the current context, we will omit the operator subscripts and simply write *rank*(), *next*() and *pre*().

Note that $pre(t_0)$ is undefined. PVS requires that all functions are total. In the case of the *pre*() operator, this is easily accomplished through the use of the subtype:

$$noninit\_elem_K := \{t_n \in clock_K | \neg init(t_n)\}$$

as the *pre*() operator's domain.

PVS allows the application of a function to any element belonging to a supertype of the function's domain and then generates a proof obligation or Type Correctness Condition (TCC). The TCC requires the user to prove the element the function is applied to is of the same type as the function's domain.

In this case, whenever the *pre*() operator is applied to an arbitrary clock value y, the pre_TCC1[1] is generated requiring the user to prove that y is never equal to 0,

[1]The authors of [4] have proved this TCC through the use of their self-defined lemma called *prod_monotonic2*. We simplify the proof based on the prelude file of PVS version 2.3, patch level 1.2.2.36

and hence has a previous value. At the same time, $y - K$ must be a sample instance, i.e., $y - K = n * K$ for some natural number n.

```
% Subtype TCC generated (at line 23, column 16) for  y - K
  % proved - complete
pre_TCC1: OBLIGATION
  FORALL (y): y - K >= 0 AND (EXISTS (n: nat): y - K = n * K);
```

We now state a definition that will aid us in defining the timing operators in the remaining subsections. For $clock_K$, the set of clock predicates, denoted $pred(clock_K)$, is the set of all boolean functions of $clock_K$:

$$pred(clock_K) := \{f | f : clock_K \rightarrow \{TRUE, FALSE\}\}$$

Figure 3.1 contains the PVS specification file that implements the parametrized theory Clocks defining the type clock that corresponds to $clock_K$ above.

```
Clocks[ K: posreal ]: THEORY
BEGIN
non_neg: TYPE = { x: real | x>=0 }
time: TYPE = non_neg
t: VAR time
clock: TYPE = { t: time|EXISTS(n:nat): t=n*K }
x: VAR clock
init(x): bool = (x=0)
noninit_elem: TYPE ={ x | not init(x) }
y: VAR noninit_elem
pre(y): clock = y - K
next(x): noninit_elem = x + K
rank(x): nat = x/K
END Clocks
```

Figure 3.1: PVS for Clocks Theory

3.1.1 General clock induction

The *general clock induction (GCI)* theorem is a simple statement of proof by general mathematical induction over clock values. It says that for a clock predicate P, if (i)

$P(t_0)$ is $true$, and (ii) for any $n > 0$, $P(t_{n-1})$ is $true$ implies that $P(t_n)$ is $true$, then $P(t_n)$ is $true$ for all t_n in $clock_K$. The following proposition is the PVS definition of the general clock induction.

```
clock_induction: PROPOSITION
  FORALL (P: pred[clock]):
    (FORALL (x: clock): init(x)
      IMPLIES P(x)) AND
    (FORALL (y: noninit_elem): P(pre(y))
      IMPLIES P(y))
      IMPLIES (FORALL (x: clock): P(x))
```

Figure 3.2: PVS proposition for general clock induction

We will use this proposition later in Chapters 4 and 5 to prove that an SRS function and SDD function are equivalent at all sample instances.

3.1.2 Strong clock induction

Frequently, to prove properties of functions defined by general inductive definitions, it is convenient to use a stronger form of mathematical induction that is called *strong induction*. Accordingly, we can specify a *strong clock induction (SCI)* in PVS to handle proofs involving clock variables. The following is the PVS definition of the *SCI*.

```
clock_strong_induction: PROPOSITION
  FORALL (P: pred[clock]):
    (FORALL (t:clock):
    (FORALL (z:clock): z < t IMPLIES P(z)) IMPLIES P(t))
      IMPLIES (FORALL (t: clock): P(t))
```

Figure 3.3: PVS proposition for strong clock induction

Here P is a clock predicate with t as a free clock variable in its domain. If for every clock value z where $z < t$, $P(z)$ is $true$ implies that $P(t)$ is $true$, then $P(t)$ is $true$ for all t in $clock_K$.

3.1.3 Comparison

The strong clock induction (SCI) can be derived from the *general clock induction (GCI)* in that *SCI* satisfies the base step and inductive step of *GCI*. In PVS, we can formally prove the proposition of *SCI* in Figure 3.1.2 through the use of *GCI* defined in Figure 3.1.1. The interested reader is referred to Appendix A.2 for details of the proof.

The benefit of the *strong clock inductions* is that one is allowed to use a broader induction hypothesis. In contrast, the general clock induction may have to take many more steps in the inductive step. We confirm this observation by proving a theorem through the use of different clock induction. The interested reader is referred to Appendix E.2 for details.

3.2 PVS Implementation of Held_For

In the underlying real-time control system, it has to decide whether a condition (e.g., the temperature is over 100 degrees) has been held for a fixed period of time (e.g., 5 minutes) and then the control system will employ the corresponding action (e.g., shutdown the power). Therefore it is necessary to define a timing operator like Held_For to handle those case. In this section, we define the PVS implementation of the HELD_FOR operator that will be used throughout the later chapters[2].

Definition 3.2.1 *Let duration denote a non-negative real number, and P represent a clock predicate. HELD_FOR is an infix operator that takes a clock predicate as its first argument, a non-negative real number as its second argument and returns a clock predicate:*

$$HELD\_FOR : pred(clock_K) \times \mathbb{R}^{\geq 0} \rightarrow pred(clock_K)$$

such that $(P)HELD\_FOR(duration)(t_n) = TRUE$ *iff*

$$(\exists t_j \in clock_K)(t_n - t_j \geq duration) \land (\forall t_i \in clock_K)(t_j \leq t_i \leq t_n \Rightarrow P(t_i))$$

[2]The current definition is a recursive one. Appendix B.3 also contains preliminary work on an alternative definition.

Here we use $\mathbb{R}^{\geq 0}$ to denote non-negative real numbers. Example 3.2.2 below illustrates how the Held_For operator works.

Example 3.2.2 *Let $K = 100$, and duration $= 195$. The clock of period* 100 *is simply*

$$clock_{100} := \{0, 100, 200, 300, \ldots\}$$

We define $Sensor(t)$ to be a clock predicate which returns true *only when $t > 50$. Figure 3.4 indicates that the smallest clock value at which the function f returns true is 300.*

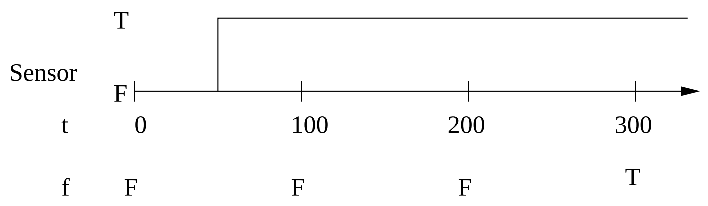


Figure 3.4: $f = (Sensor)$HELD_FOR(195) example

Note that we are ignoring the inter-sample behavior of $Sensor$. The truth value of HELD_FOR is only dependent upon the value of $Sensor$ at the sampling instances corresponding to the clock values.

3.2.1 The recursive definition in PVS

The PVS theory defining the recursive version of the Held_For operator is shown in Figure 3.5.

The PVS function implementing the HELD_FOR operator is $Held\_For$, defined at the bottom of the theory. It implements the HELD_FOR operator by evaluating the recursive function *heldfor*, which, as long as $P(t)$ is $true$, back tracks to the previous value of t until $t\_now - t$ is greater than or equal to *duration*. If at any point before the recursion terminates $P(t)$ is $false$ or the initial state is reached, *heldfor* returns $false$.

```
Held_For  [K:posreal] : THEORY
  BEGIN
  IMPORTING Clocks[K]
  t, t_now: VAR clock
  duration:VAR time
  P: VAR pred[clock]
  heldfor(P, t, t_now, duration):
    RECURSIVE bool =
        IF P(t) THEN
          IF (t_now - t >= duration) THEN TRUE
          ELSIF init(t) THEN FALSE
          ELSE heldfor(P,pre(t),t_now,duration)
          ENDIF
        ELSE FALSE
        ENDIF
        MEASURE rank(t)
   Held_For(P, duration): pred[clock] =
    (LAMBDA (t:clock): heldfor(P,t,t,duration))
   END Held_For
```

Figure 3.5: PVS recursive definition of HELD_FOR operator

The function `heldfor` involves the PVS recursive definition. According to the PVS convention, if a function is recursive it must be declared as such with the keyword `RECURSIVE` and given a `measure` function and an order relation (e.g., <) to guarantee it terminate. When this is typechecked a TCC is generated requiring the user to prove that the `measure` strictly decreases with respect to the given order with every step in the recursion. In this function, the order relation is `<` on `nat`. As stated earlier, the function `rank` maps a clock value to a natural number.

The recursive definition of the Held_For operator can significantly reduce the manual intervention in the actual theorem proof. The major problem of this definition is that "unspooling" of the recursive definition is required during the proof. It is therefore not practical to verify timing blocks that have a large delay relative to the system loop time.

The theory `simple` in Figure 3.6 illustrates the use of the PVS Held_For theory. The theorem `Good_held_for` is easily proved by the PVS `(GRIND)` command, which

```
  simple  : THEORY
  BEGIN
  K: posreal = 100
  IMPORTING Held_For[K]
  t, t1, t2: VAR clock
  param_trip: timed_cond = (LAMBDA (t):
       IF (t<1000) THEN FALSE ELSE TRUE ENDIF)

  duration:posreal = 295
  clock_lt_le: lemma t1 < t2 => t1 <= t2 - K

  Good_held_for: THEOREM (t>=1000+duration) IMPLIES
          Held_For(param_trip,duration)(t)

  Bad_held_for: THEOREM (t>=1000+duration-K) IMPLIES
Held_For(param_trip,duration)(t)

  Not_bad_held_for: THEOREM  EXISTS (t: clock[K]):
 not
  ((t>=1000+duration-K) IMPLIES
     Held_For(param_trip,duration)(t))
   END simple
```

Figure 3.6: PVS file utilizing Held_For

tries repeated skolemization, instantiation and if-lifting to simplify the sequent(s). This result is expected since the first clock value greater than $1000 + duration$ is 1300 and in this case $1300 - 1000 > 295$ while *param_trip* is true at 1000, 1300 and all clock values in between.

Attempting to prove the theorem `Bad_held_for` results in the unprovable sequent:

```
Bad_held_for.1 :
[-1]    n!1 >= 0
[-2]    100 * n!1 >= 0
[-3]    t!1 = 100 * n!1
[-4]    (100 * n!1 >= 1195)
  |-------
{1}   param_trip(100 * n!1 - 300)
```

This unprovable sequent corresponds to the equation:

$$(\forall t_n \in clock_{100}) t_n \geq 1195 \Rightarrow param\_trip(t_n - 300)$$

The number $1195 = 1000 + 295 - 100 = 1000 + duration - K$. The first clock value greater than or equal to this number is 1200, but when t!1=1200 all formulas are true except {1} since *param_trip(900) = FALSE* resulting in an unprovable sequent.

For more complex examples it is difficult, if not impossible to tell whether the theorem as stated is false, and hence unprovable, or if we simply are not being clever enough in the way we use PVS. In an effort to clarify this situation we can perform "refutation" using PVS. The theorem of `Not_bad_held_for` is an example of "refutation", which negates the theorem `Bad_held_for`. Since the theorem `Not_bad_held_for` has been proved using PVS, we have confirmed that `Bad_held_for` is unprovable.

3.3 PVS Lemmas for Timing Properties

In the proof of the theorem `set_implies_recursive` in Appendix B.2 we employ an instance of a user-defined lemma named `lt_le_clockvar`. In PVS, axioms, assumption, or previously proved results can be seen as lemmas introduced by the command (`lemma` *names*). In order to maintain the PVS system consistency, we try to avoid using axioms and assumption in the PVS specification file. Once a lemma is introduced, this lemma together with associated TCCs should be been proved using low level PVS operations or lemmas from the PVS prelude file.

3.3.1 Lemmas "pulled out" from the theorem proof

In the previous example, `lt_le_clockvar` is used to prove the following sequent:

```
[-1]    t!2 < y!1
  |-------
[1]    (t!2 <= y!1 - K)
```

Here t!2 and y!1 are clock variables. Now we use low level PVS commands instead of lemma `lt_le_clockvar` to prove this sequent.

First of all, we use `(TYPEPRED "t!2" "y!1")` to add type constraints for t!2 and y!1

```
{-1}    t!2 >= 0
{-2}    EXISTS (n: nat): t!2 = n * K
{-3}    y!1 >= 0
{-4}    EXISTS (n: nat): y!1 = n * K
[-5]    t!2 < y!1
  |-------
{1}    init[K](y!1)
[2]    (t!2 <= y!1 - K)
```

Second of all, we try repeated skolemization, instantiation, and if-lifting by using `(GRIND)`. The sequent simplifies to:

```
{-1}    n!2 >= 0
{-2}    n!1 >= 0
{-3}    n!1 * K >= 0
{-4}    t!2 = n!1 * K
{-5}    n!2 * K >= 0
{-6}    y!1 = n!2 * K
{-7}    n!1 * K < n!2 * K
  |-------
{1}    init[K](n!2 * K)
{2}    (n!1 * K <= n!2 * K - K)
```

Then use `(BOTH-SIDES "/" "K" -7)` and `(BOTH-SIDES "/" "K" 2)` to have both sides of {-7} and {2} divided by K.

```
[-1]    n!2 >= 0
[-2]    n!1 >= 0
[-3]    n!1 * K >= 0
[-4]    t!2 = n!1 * K
[-5]    n!2 * K >= 0
[-6]    y!1 = n!2 * K
[-7]    n!1 * K / K < n!2 * K / K
  |-------
[1]    init[K](n!2 * K)
{2}    n!1 * K / K <= (n!2 * K - K) / K
```

Finally we use `(GRIND)` again to complete the proof.

It is very often the case that some of the proof patterns, like the above, are used repeatedly in many different theorem proofs, or even within one theorem proof. To reduce the required effort and optimize the proof, we represent the repeated pattern in a user-defined lemma. Here `lt_le_clockvar` is an example of this kind of a user-defined lemma which is "pulled out" from the actual theorem proof. The use of user-defined lemmas help to enrich the PVS real-time library.

In the following subsections, we list some of the important lemmas and provide brief explanations of their use, including lemmas about inter-sample time values, lemmas about Held_For operator and lemmas about the loop time K and duration of the Held_For operator.

3.3.2 Lemmas about inter-sample time values

As mentioned earlier, we are ignoring inter-sample behavior of inputs. Sometimes, however, we have to compare the clock values (sample instance) with non-sample instance. For example, a *duration* is a non-negative real number and is not necessarily an integer multiple of clock period K. It is often necessary to relate the non sample instance to the clock values. This is accomplished by following lemmas:

```
t1, t2: var clock
le_inner_clockvar: lemma
     forall (x:{y: posreal|(t1<y) & (y<t1+K)}): (t2<=x) => (t2<=t1)

ge_inner_clockvar: lemma
     forall (x:{y: posreal|(t1<y) & (y<t1+K)}): (t2>=x) => (t2>=t1+K)
```

Here x is a positive real number in between two consecutive clock values, i.e., $t1 < x < next(t1)$. The above two lemmas state that for any clock variables $t1$ and $t2$, if $t2 \leq x$ then $t2 \leq t1$; if $t2 \geq x$ then $t2 \geq next(t1)$.

```
lt_le_clockvar: LEMMA t1 < t2 => t1  <= pre(t2)

gt_ge_clockvar: LEMMA t1 > t2 => t1 >= next(t2)
```

In above lemmas $t1$ and $t2$ are clock variables. For any t1, t2, if $t1 < t2$ then $t1 \leq pre(t2)$; if $t1 > t2$ then $t1 \geq next(t2)$.

```
eq_inner_clockvar: lemma  forall (t1, t2: clock):
       (t1-K  <=t2 and t2 <=t1) implies
            t2=t1-K or t2 = t1

eq_inner3_clockvar: lemma forall (t1, t2):
       (t1-2*K  <=t2 and t2 <=t1) implies
         t2=t1-2*K or  t2=t1-K or t2 = t1

 zero_inner_clockvar: lemma
       forall (t: clock): t<K => t=0
```

The above three lemmas are used to locate a clock variable $t2$ which is in between 2 or more consecutive clock values.

3.3.3 Lemmas about loop time K and Held_For duration

In the following two lemmas, the *looptimeK* is positive real number and *duration* is non-negative real number.

```
loop_duration1: lemma FORALL (n: {n: nat |n /= 0}):
       K >= duration IMPLIES
           n*K >= duration
```

If $K \geq duration$ then $n * loop\_time \geq duration$ if n is positive natural number.

```
loop_duration2: lemma EXISTS n: n*K >= duration
```

For any arbitrary value of K and *duration*, we can always find a natural number n such that $n * K \geq duration$.

Chapter 4

Verification of the Channel Trip Sealed-in Block

This chapter performs the verification of the channel trip reset block of an industrial control system through the use of the *PVS-RT* method. The original Software Requirement Specification (SRS) and the Software Design Description (SDD) for this timing block are represented in the PVS specification language. We then specify the main proof obligation of the timing block comparison according to the decomposed 4-variable model in Section 2.5. Attempting to prove this main proof obligation results in several unprovable sequents, which are used to diagnose errors in both the specification and the implementation. Finally we provide modified SRS and SDD definitions where all of the detected errors have been fixed and all the proof obligations been discharged successfully. Thus the modified SRS specification and SDD implementation have equivalent functionality for all the input values and the system engineers will have a higher degree of confidence in the system's correctness.

4.1 Overview

The industrial control system under examination is a watchdog system. The watchdog system is a "poised" system that is not called upon to operate in normal conditions but rather monitors the plant parameters and reacts to shutdown or "trip" the system only if anomalous behavior is observed for one or more of the plant parameters. When such anomalous behavior is exhibited by a plant parameter we say that there is a "parameter trip". Parameter trip is first computed by the watchdog system and then used to determine if there should be a "channel trip" to indicate that the plant should be shut down.

As shown in Figure 4.1, this block takes two boolean valued inputs, *param_trip* and *manual_reset*, and produces a single boolean valued output *c_sealed_in* for every $K = 100$ ms. When the value of *param_trip* is continuously *true* for *seal_in_delay* = 150 ms or longer, then the channel trip is sealed and *c_sealed_in* is set to *true*. If *param_trip* lasts less than the *sealed_in_delay* time, the trip computer does not seal-in the channel trip. A sealed-in channel trip can be cleared only by a manual reset indicated by making *manual_reset* = *true*.

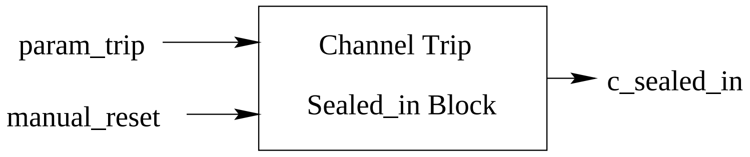


Figure 4.1: Block diagram for channel trip sealed-in

4.2 The Original Definitions

In this section, we provide the original Software Requirement Specification (SRS) and the Software Design Description (SDD) for the channel trip sealed-in function. Both are presented as tabular expressions with separate descriptions of initial conditions. The PVS versions of the definitions immediately follow the original definitions.

4.2.1 The SRS specification

The complete specification and design require fail-safe operation so the value of the block output *c_sealed_in* is initialized to *true*. For any non-initial lock value, the result of *c_sealed_in* is defined in Table 4.1

As illustrated in Section 2.4.2, Table 4.1 is a complex horizontal condition table with labels of *Condition* and *Result*. The inputs to this SRS function are *param_trip*

Table 4.1: The original SRS for the channel trip sealed-in block

Condition		*Result* c_sealed_in
(param_trip) Held for (seal_in_delay)		true
NOT [(param_trip) Held for (seal_in_delay)]	manual_reset	false
	¬manual_reset	No Change

and *manual_reset*, both of which are of type *pred*[*clock*]. The function output is a boolean variable. When the conjunction of atomic propositions in a given row of the *Condition* columns is *true*, then *param_trip* is set to the *Result* value for that row. For example, when

$$\neg\ [(param\_trip) Held\_For(seal\_in\_delay)] \wedge manual\_reset \text{ is true}$$

then $c\_sealed\_in =$ false.

The PVS version of the SRS definition is shown as follows:

```
c_sealed_in(param_trip,manual_reset)(t): RECURSIVE bool =
IF  init(t) THEN  TRUE ELSE
 COND
    Held_For(param_trip,seal_in_delay)(t) -> TRUE,
    NOT Held_For(param_trip,seal_in_delay)(t) AND
       manual_reset(t) -> FALSE,
    NOT Held_For(param_trip,seal_in_delay)(t) AND
       NOT manual_reset(t) ->
         c_sealed_in(param_trip,manual_reset)(pre(t)),
   ENDCOND
 ENDIF
MEASURE rank(t)
```

4.2.2 The SDD implementation

In the Software Design Description (SDD), Table 4.2 defines the state transition function `ECHNT` for non-initial clock values.

In this table, the input variables are *param_trip*, *manual_reset*, $l\_SealDly_{-1}$ and $CHNTS_{-1}$. The first two are of the type *pred*[*clock*] as those in the SRS function.

Table 4.2: State transition function ECHNT

			Results	
Condition			CHNTS	l_SealDly
NOT $param\_trip$	$CHNTS_{-1}$ =SEAL	$manual\_reset$	NCHNT	0
		$\neg manual\_reset$	SEAL	0
	$CHNTS_{-1} \neq$ SEAL		NCHNT	0
$param\_trip$	$l\_SealDly_{-1}$=0		CHNT	next(0)
	$0 < l\_SealDly_{-1} < seal\_in\_delay$		NC	next($l\_SealDly_{-1}$)
	$l\_SealDly_{-1} \geq seal\_in\_delay$		Seal	0

The last two variables are the system feedback inputs where the subscript -1 denotes the previous state value.

There are two columns under the label *Results*, indicating that the defined function returns two values, *CHNTS* and *l_SealDly*. Here *l_SealDly* is the clock value of a timer which is used to implement the Held_For operator employed by the SRS.

The other returned value $CHNTS$ is a three valued function so that this output could also provide some information about $param\_trip$ to the rest of the system. In this table, $CHNTS_{-1}$ denotes the value of $CHNTS$ at the previous state. The idea is that $CHNTS = SEAL$ when the channel trip is sealed; it is $CHNT$ when the channel trip is unsealed and $param\_trip = true$; it is $NCHNT$ when the channel trip is unsealed and $param\_trip = false$. The value NC in the $CHNTS$ column is short for "No Change".

Using the above information, Table 4.2 can be translated into the following PVS function:

```
KSEAL: nat = 150
Channel_state: TYPE = {NCHNT, CHNT, SEAL}
SDD_State: TYPE = [# CHNTS1: Channel_state, l_SealDly: clock#]

ECHNT(PT:bool,MR:bool,l_SealDly_s1:clock,CHNTS_s1:Channel_state): SDD_State = COND
  NOT PT AND CHNTS_s1=SEAL AND MR ->
        (# CHNTS:= NCHNT, l_SealDly:= 0 #),
  NOT PT AND CHNTS_s1=SEAL AND NOT MR ->
        (# CHNTS:= SEAL, l_SealDly:= 0 #),
  NOT PT AND NOT CHNTS_s1=SEAL ->
```

```
        (# CHNTS:= NCHNT, l_SealDly:= 0 #),
    PT AND l_SealDly_s1=0     ->
        (# CHNTS:= CHNT, l_SealDly:= next(0) #),
    PT AND 0<l_SealDly_s1 AND l_SealDly_s1<KSEAL ->
        (# CHNTS:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
    PT AND l_SealDly_s1>=KSEAL ->
        (# CHNTS:= SEAL, l_SealDly:= 0 #)
    ENDCOND
```

Here the boolean variables PT and MR denote the parameter trip and manual reset, respectively. We use *l_SealDly_s1* and *CHNTS_s1* to represent $l\_SealDly_{-1}$ and $CHNTS_{-1}$ in Table 4.2. This function returns a two-valued output of type `SDD_State`, which can be declared in PVS as follows:

$$\texttt{SDD\_State} : \texttt{TYPE} = [\#\texttt{CHNTS} : \texttt{Channel\_state}, \texttt{l\_SealDly} : \texttt{clock}\#]$$

In PVS, this kind of type is called a *record type*. A *record type* has some *record accessors* or *fields*, e.g., *CHNTS* and *l_SealDly* in this case. The interested reader is referred to Section 4.4 of [16] for further details.

On initialization CHNTS is set to $CHNT$ and the timer *l_SealDly* is set to 0. Taking the initial values into consideration, we obtain the complete SDD function in PVS:

```
% Declare constants for the initial state
ICHNT: SDD_State = (# CHNTS1:= CHNT, l_SealDly:= 0 #)

% main SDD function
sdd(param_trip,manual_reset)(t): RECURSIVE SDD_State = COND
   init(t) -> ICHNT,
   not init(t) -> ECHNT(param_trip(t), manual_reset(t),
  l_SealDly(sdd(param_trip,manual_reset)(pre(t))),
  CHNTS1(sdd(param_trip,manual_reset)(pre(t))))
   ENDCOND
   MEASURE rank(t)
```

4.3 Analysis of the Verification Results

This section first presents the block comparison theorem. Our attention will then focus on the unprovable sequents generated when the theorem proof fails. These unprovable sequents lead to the discovery of errors in both the original specification and the proposed implementation.

4.3.1 Proof obligations

To ensure the SDD implement the SRS specification, we have to discharge two kinds of the proof obligation in PVS:

(1) The *Type Correctness Conditions (TCCs)* generated by PVS.

(2) The main block comparison theorem: $Abst_C \circ REQ = SOF_{req} \circ Abst_M$ as described in Section 2.5.2.

In this timing block, all the TCCs have been discharged either by the PVS built-in automatic proof strategy or by manual proof. So we will focus our attention on the proof obligation (2). The proof obligation (2) results in the following PVS theorem

```
Seal: THEOREM
 c_sealed_in(param_trip,manual_reset)(t) =
       SEAL?(CHNTS1(sdd(param_trip,manual_reset)(t)))
```

As illustrated earlier, the function `sdd` returns a two field record expression. It is very often the case that one *field* of a record expression is referenced. PVS provides associated record access function. For instance, the expression

$$\texttt{CHNTS(sdd(param\_trip, manual\_reset)(t)))}$$

indicates that the value of $CHNTS$ has been selected. In the above theorem specification, the SRS function `c_sealed_in` returns a boolean value, whereas the SDD function could returns one of three state values, i.e., $SEAL$, $NCHNT$ or $CHNT$. When `c_sealed_in` is $true$, the function `sdd` must returns $SEAL$; if `c_sealed_in` is $false$, the `sdd` may returns either $CHNT$ or $NCHNT$. In PVS, we use

$$\texttt{SEAL?(CHNTS1(sdd(param\_trip, manual\_reset)(t)))}$$

to represent

$$\mathtt{CHNTS1}(\mathtt{sdd}(\mathtt{param\_trip}, \mathtt{manual\_reset})(\mathtt{t})) = \mathtt{SEAL}$$

Figure 4.2 is a commutative diagram based on the main block comparison theorem `Seal`. As we can see, the function `c_sealed_in` corresponds to REQ and the function `sdd` corresponds to SOF_{req}. Since the function arguments for the SRS and SDD are the same, i.e., (pred[clock], pred[clock], clock), the abstraction function $Abst_{\mathbf{M}}$ can be viewed as the identity maps on these three types of variables, i.e.,

$$Abst_{\mathbf{M}} = id_{pred[clock]} \times id_{pred[clock]} \times id_{clock}$$

The other abstraction function $Abst_{\mathbf{C}}$ maps the controlled variables in the SRS to the SDD, i.e., $Abst_{\mathbf{C}} : \mathbf{C_{SRS}} \rightarrow \mathbf{C_{SDD}}$. In this timing block, $Abst_{\mathbf{C}}$ = SEAL?

$$\begin{tikzcd}
M_{SRS} \arrow[r, "c\_sealed\_in"] \arrow[d, "Abst_M"] & C_{SRS} \\
M_{SDD} \arrow[r, "CHNTS1(sdd(..))"'] & C_{SDD} \arrow[u, "Abst_C"']
\end{tikzcd}$$

Figure 4.2: Commutative diagram of the block comparison theorem

To apply PVS to this verification problem, we use the strategy `(INDUCT "t" 1 "clock_induction")`. This breaks the proof into two parts: (i) Base Case when $t = 0$, and (ii) Inductive Case. In the course of proving these cases, we find the following errors regarding the original implementation.

1. Wrong initial condition for CHNTS.

2. Wrong initial condition for l_SealDly.

3. CHNTS becomes unlocked without a manual reset.

4. Manual reset unseals the SRS channel but not SDD channel.

5. Incorrect transition when manual reset is deactivated during the parameter trip.

In the following subsections, we will discuss these errors in detail as they are discovered through the analysis of the unprovable sequents.

4.3.2 Wrong initial condition for CHNTS

As mentioned earlier, the value of *c_sealed_in* in the SRS was initially set to *true* so that any failure leaves the system in a safe condition. However, in the original design of the SDD, $CHNTS$ is initialized to $NCHNT$, rather than $SEAL$. So the following unprovable sequent is encountered when we apply the *clock induction* in the proof of the main theorem `Seal`.

```
[-1]    (x!1 = 0)
  |-------
[1]    SEAL?(CHNT)
```

The above sequent shows that if the system is in the initial state ($x!1 = 0$), then $CHNT$ equals to $SEAL$ (i.e., SEAL?(CHNT)). We have no way to prove this sequent.

To solve this problem, we need to change the initial value of CHNTS to $SEAL$.

4.3.3 Wrong initial condition for l_SealDly

At the initial state ($t = 0$), the timer value *l_SealDly* is set to 0. When there is no a parameter trip at $t = 0$, this assignment is fine . However, if a parameter trip exists in the initial state, this assignment will result in different behaviors of the SRS and SDD.

This problem came to our attention when we failed to prove the theorem `Seal`. The following is an unprovable sequent that occurred during the proof:

```
Seal.2.3.2.1.1.1 :

[-1]    param_trip!1(0)
```

```
[-2]    manual_reset!1(100)
[-3]    param_trip!1(200)
[-4]    param_trip!1(100)
  |-------
```

From this unprovable sequent, we can get a counter example as follows

$$param\_trip!1(200) \wedge param\_trip!1(100) \wedge param\_trip!1(0) \wedge manual\_reset!1(100)$$

If the param_trip!1 has been $true$ when $t_0 = 0, t_1 = 100$ and $t_2 = 200$ and manual_reset is set to $true$ when $t_1 = 100$, then the SRS function *c_sealed_in* returns $true$ when $t_2 = 200$. Under the same condition, however, SEAL?(CHNTS1(sdd(param_trip!1, manual_reset!1)(200))) returns $false$. Figure 4.3 shows the problematic sequent.

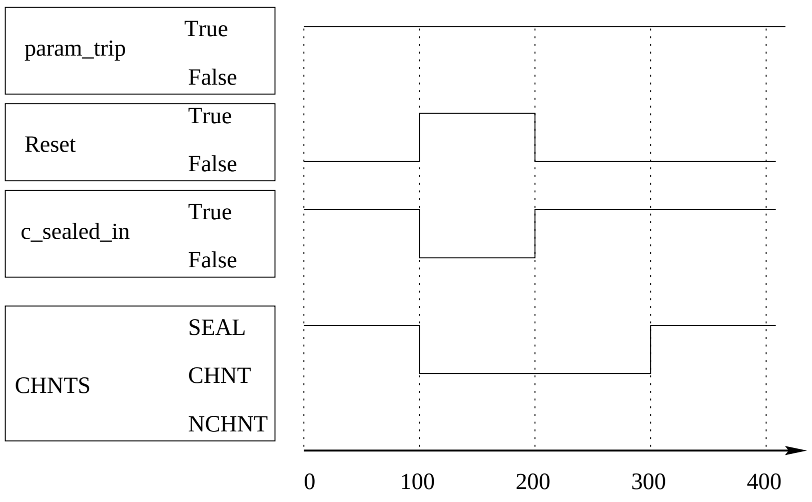


Figure 4.3: The problem of the initial l_SealDly

Furthermore, we find this unprovable sequent resulted from the initial condition of *l_SealDly*. Since *l_SealDly* is initialized to 0, the timer starts to count the parameter trip delay time (*l_SealDly*) when $t_1 = 100$, rather than $t_0 = 0$. Although the parameter trip is $true$ when $t_0 = 0$, $t_1 = 100$ and $t_2 = 200$, it is only when $t_3 = 300$ that *l_SealDly* becomes greater than $K\_SealDly(150)$ and hence $CHNTS$ becomes $SEAL$; while the SRS function `c_sealed_in` has already been $true$ at the previous state ($t_2 = 200$).

In order to fix this problem, we have to reconsider the initial value of *l_SealDly*. That is, if there is a parameter trip when $t = 0$, then $l\_SealDly$ is $next(0)$, indicating the timer begin to count the trip delay; if a parameter trip is not held $true$ at the initial state, then *l_SealDly* is set to 0. The following is the modified part of the SDD function in PVS.

```
    init(t) AND NOT param_trip(t) -> (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    init(t) AND param_trip(t)     -> (# CHNTS1:= SEAL, l_SealDly:= next(0) #),
```

4.3.4 CHNTS becomes unlocked without a manual reset

After fixing the errors resulting from the incorrect initial condition as described in the previous subsections, the main block comparison theorem `Seal` still cannot be discharged. The following is one of the unprovable sequents generated when we try to prove the theorem `Seal`:

```
Seal.2.1.3.2.1.2.2.1 :

[-1]    param_trip!1(y!1 - 300)
[-2]    param_trip!1(y!1 - 400)
[-3]    param_trip!1(y!1 - 500)
{-4}    param_trip!1(y!1 - 200)
[-5]    param_trip!1(y!1)
  |-------
[1]    manual_reset!1(y!1 - 200)
[2]    param_trip!1(y!1 - 100)
{3}    l_SealDly(sdd(param_trip!1, manual_reset!1)(y!1 - 300)) = 0
{4}    l_SealDly(sdd(param_trip!1, manual_reset!1)(y!1 - 300)) < 150
[5]    manual_reset!1(y!1 - 100)
[6]   manual_reset!1(y!1)
```

Figure 4.4 shows the different behavior of the SRS specification and the SDD implementation based on this problematic sequent.

As we see, both the SRS and SDD become sealed at time $t = y - 300$. After that, the parameter trip continues and the SRS remains sealed (i.e., c_sealed_in = True). But according to Table 4.2, the SDD becomes unsealed at the next state because the $l\_SealDly$ counter is reset to 0 when CHNTS is set to $SEAL$. As a result the system

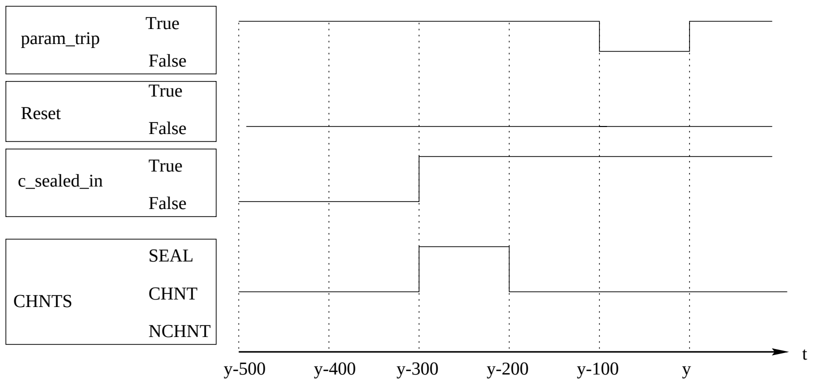


Figure 4.4: CHNTS becomes unseal without a manual reset

loses the “history” of *param_trip* and CHNTS cannot correctly represent the actual state transition.

To fix this problem, we have to keep the *l_SealDly* unchanged when CHNTS becomes *SEAL*.

So far all of the detected errors are associated with the SDD definition. The following are the revised PVS functions of the SDD. To distinguish them from the original versions, we use `ECHNT2` and `sdd2` as to the new SDD functions.

```
ECHNT2(PT:bool,MR:bool,l_SealDly_s1:clock,CHNTS1_s1:Channel_state): SDD_State =
 COND
   NOT PT AND CHNTS1_s1=SEAL AND MR ->
         (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
   NOT PT AND CHNTS1_s1=SEAL AND NOT MR ->
         (# CHNTS1:= SEAL, l_SealDly:= 0 #),
   NOT PT AND NOT CHNTS1_s1=SEAL ->
         (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
   PT AND l_SealDly_s1=0 ->
         (# CHNTS1:= CHNT, l_SealDly:= next(0) #),
   PT AND 0<l_SealDly_s1 AND  l_SealDly_s1<KSEAL ->
         (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
   PT AND l_SealDly_s1>=KSEAL ->
         (# CHNTS1:= SEAL, l_SealDly:= next(l_SealDly_s1)#)
ENDCOND
```

```
 sdd2(param_trip,manual_reset)(t): RECURSIVE SDD_State =
  COND
    init(t)  AND NOT param_trip (t) ->
          (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    init(t)  AND param_trip (t) ->
          (# CHNTS1:= SEAL, l_SealDly:= next(0) #),
    NOT init(t)  ->
          ECHNT2(param_trip(t), manual_reset(t),
 l_SealDly(sdd2(param_trip,manual_reset)(pre(t))),
 CHNTS1(sdd2(param_trip,manual_reset)(pre(t))))
  ENDCOND
  MEASURE rank(t)
```

4.3.5 The manual reset unseals the SRS but not SDD

The main block comparison theorem still cannot be proved even based on the modified SDD functions. The following is one of the unprovable sequents.

```
Seal2.2.2.1.1 :
[-1]    manual_reset!1(100)
[-2]    param_trip!1(100)
[-3]    param_trip!1(0)
  |-------
```

From this unprovable sequent, we can derive a counter example as follows:

$$manual\_reset!1(100) \wedge param\_trip!1(100) \wedge param\_trip!1(0)$$

When the param_trip has been true when$t_0 = 0$ and $t_1 = 100$ and the manual_reset is activated at time $t_1 = 100$, the SRS function `c_sealedin` is unsealed when $t_1 = 100$. However, the SDD function `sdd2` is in state $SEAL$ when $t_1 = 100$ so the channel is still sealed. Figure 4.5 shows the different behavior of the SRS and SDD.

Compared to the original SRS specification of Table 4.1, the SDD definition of Table 4.2 does not allow a manual reset to affect the sealed-in trip when there is a parameter trip. To solve this problem, we have to specify that if

$$\neg init(t) \wedge PT \wedge (l\_SealDly_{-1} < sealed\_in\_delay) \wedge MR$$

then CHNTS returns $CHNT$ and $l\_SealDly$ returns $next(l\_SealDlyz_{-1})$.

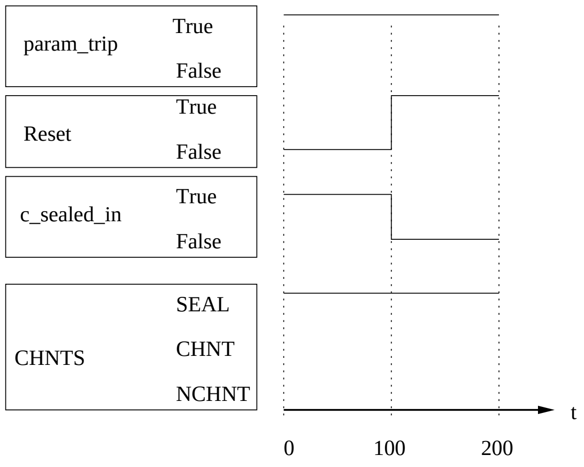


Figure 4.5: Manual_reset unseals the SRS but not SDD

4.3.6 Incorrect transition when a manual reset is deactivated during the parameter trip

In the previous subsection, we discuss the case when a *manualreset* happens during the parameter trip. The following unprovable sequent then force us to review the specified transition when a *manual_reset* is deactivated during the parameter trip.

```
Seal2.2.1.2.1.1.1 :
[-1]   param_trip!1(100)
  |-------
[1]    manual_reset!1(100)
[2]    param_trip!1(0)
```

From this sequent, we know that the SRS is not equivalent to the SDD when

$$\neg param\_trip!1(0) \wedge param\_trip!1(100) \wedge \neg manual\_reset!1(100) \text{ is true}$$

Figure 4.6 shows that when the input sequence is

$$\neg param\_trip!1(100) \wedge param\_trip!1(100) \wedge \neg manual\_reset!1(100)$$

The SRS channel is sealed (`c_sealed_in` becomes true) when $t = 100$ while CHNTS changes from $SEAL$ to $CHNT$ and hence the SDD channel is unsealed at that time.

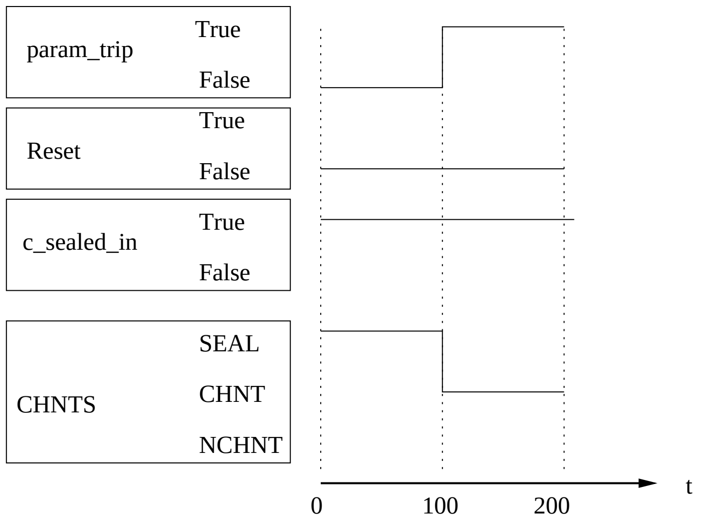


Figure 4.6: Incorrect transition when manual_reset is false during the parameter trip

To fix this problem, we have to specify that under the condition that

$$\neg init(t) \wedge PT \wedge l\_SealDly_{-1} < sealed\_in\_delay \wedge \neg MR,$$

if $CHNTS_{-1}$ is $SEAL$ then $CHNTS$ remains unchanged; if $CHNTS_{-1} \neq SEAL)$, $CHNTS$ goes to $CHNT$. In both case, $l\_SealDly$ returns $next(l\_SealDly_{-1}$.

4.4 The "Fixed" Definitions

In this section we provide the "fixed" version of the SDD below, where all of the detected errors have been corrected.

Since the transitions regarding the timer counter $l\_SealDly$ exists at the initial state, we include them in Table 4.3. Here PT and MR are short for the parameter trip and manual reset, respectively.

Table 4.3: The fixed version of the SDD

					Results	
Condition					CHNTS1	l_SealDly
init(t)	PT				SEAL	next (0)
	¬ PT				SEAL	0
¬ init(t)	¬ PT	$CHNTS1_{-1}$ =SEAL	MR		NCHNT	0
			¬MR		SEAL	0
		$CHNTS1_{-1}$ ≠SEAL			NCHNT	0
	PT	$l\_SealDly_{-1}$ < $seal\_in\_delay$	MR		CHNT	next($l\_SealDly_{-1}$)
			¬ MR	$CHNTS1_{-1}$ ≠SEAL	CHNT	next($l\_SealDly_{-1}$)
				$CHNTS1_{-1}$=SEAL	SEAL	next($l\_SealDly_{-1}$)
		$l\_SealDly_{-1} \geq seal\_in\_delay$			SEAL	NC

To accord with the changes in Table 4.3, the SDD function in PVS has been revised. The following are the third version of the SDD functions labeled `ECHNT3` and `sdd3`. The main block comparison theorem based on the newest version has been proved successfully. The interested reader is referred to Appendix E.2 for the detailed proof strategy.

```
ECHNT3(PT:bool,MR:bool,l_SealDly_s1:clock,CHNTS1_s1:Channel_state): SDD_State =
 COND
   NOT PT AND CHNTS1_s1=SEAL AND MR ->
         (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
   NOT PT AND CHNTS1_s1=SEAL AND NOT MR ->
         (# CHNTS1:= SEAL, l_SealDly:= 0 #),
   NOT PT AND NOT CHNTS1_s1=SEAL ->
         (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
   PT AND l_SealDly_s1 < KSEAL AND MR  ->
         (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
   PT AND l_SealDly_s1 < KSEAL AND
        NOT MR AND  CHNTS1_s1=SEAL ->
         (# CHNTS1:= SEAL, l_SealDly:= next(l_SealDly_s1) #),
   PT AND l_SealDly_s1<KSEAL AND NOT MR AND NOT CHNTS1_s1 = SEAL ->
         (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
   PT AND l_SealDly_s1>=KSEAL ->
```

```
         (# CHNTS1:= SEAL, l_SealDly:= next(l_SealDly_s1)#)
 ENDCOND

sdd3(param_trip,manual_reset)(t): RECURSIVE SDD_State =
 COND
   init(t)  AND NOT param_trip (t) ->
         (# CHNTS1:= SEAL, l_SealDly:= 0 #),
   init(t)   AND     param_trip (t) ->
         (# CHNTS1:= SEAL, l_SealDly:= next(0) #),
   NOT init(t)  ->
          ECHNT3(param_trip(t), manual_reset(t),
 l_SealDly(sdd3(param_trip,manual_reset)(pre(t))),
CHNTS1(sdd3(param_trip,manual_reset)(pre(t))))
 ENDCOND
MEASURE rank(t)
```

4.5 The "safer" Versions of the SRS and SDD

The PVS verification results show that the modified version of the SDD implements the SRS requirements successfully. However the experts working on this industrial control system insist that the specification and implementation could be made "safer". The additional "safer" requirement is:

> *A manual reset should only be allowed if the parameter is not currently tripped.*

To accord with this requirement, Table 4.4 and 4.4 revise the "fixed" definitions of the SRS and SDD by showing that the manual reset will not alter the channel sealed-in trip if a parameter trip was true in the recent past.

In Appendix E.1, we provide the "safer" version of the PVS specification theory named `sealed_in_safe`. The main block theorem based on this version has been proved successfully.

Table 4.4: The "safer" version of the SRS

Condition			*Result* param_trip
(param_trip) Held for (seal_in_delay)			True
NOT [(param_trip) Held for (seal_in_delay)]	Manual Reset	¬param_trip	False
		param_trip	No Change
	¬Manual Reset		No Change

Table 4.5: The "safer" version of the SDD

Condition				*Results* CHNTS1	l_SealDly
init(t)	PT			SEAL	next (0)
	¬ PT			SEAL	0
¬ init(t)	¬ PT	$CHNTS1_{-1}$ =SEAL	MR	NCHNT	0
			¬ MR	SEAL	0
		$CHNTS1_{-1} \neq$ SEAL		NCHNT	0
	PT	$l\_SealDly_{-1} <$ $seal\_in\_delay$	$CHNTS1_{-1} \neq$SEAL	CHNT	next($l\_SealDly_{-1}$)
			$CHNTS1_{-1}$=SEAL	SEAL	next($l\_SealDly_{-1}$)
		$l\_SealDly_{-1} \geq seal\_in\_delay$		SEAL	NC

4.6 Summary

In this chapter, we first provide the original specification and the implementation of the channel trip sealed-in block in PVS specification language. Based on these definitions, we try to prove that the SRS specification and SDD implementation have equivalent functionality for all the input values. Our attempts failed and result in some unprovable sequents. Analyzing the unprovable sequents leads to finding 5 implementation errors in the original definition. The original SRS specification is also less safer so we make some adjustments in the SRS and SDD. After fixing those errors, we provide the updated version of the SRS and SDD and prove all of the TCCs and the main block comparison theorem.

Chapter 5

Verification of the Channel Reset Block

The channel trip sealed-in block is another subsystem of the industrial real-time control system which we first encountered in Chapter 4. It determines the channel reset status depending on the input reset switch status. In this chapter we begin by providing the original SRS specification and proposed SDD implementation. The PVS code defining the block comparison is then given and the proof techniques for the TCCs and the main block comparison theorem are described. The resulting failed proofs are used to diagnose the source of the failure. At the end, we provide the modified SRS and SDD definitions where all of the detected errors have been fixed and all the proof obligations have been discharged.

5.1 Overview

As stated in Chapter 4, the tripped (sealed) channel will be reset through the use of the manual reset switch. The reset switch has a binary value (*On* or *Off*). Turning on (or off) the switch may have 100 ms debounce time. If the switch has been turned on for 200 ms, it is regarded as jammed or stuck. Depending on value of the reset switch and how long this value has been held, the channel reset state could be:

(1) waiting for the reset switch *on* ($wait\_for\_on$),

(2) waiting for the reset switch *off* ($wait\_for\_off$),

(3) reset switch getting stuck (*stuck*), or

(4) trip being clear (*cleartrip*).

The block diagram for this timing block is shown in Figure 5.1. As we see, the block

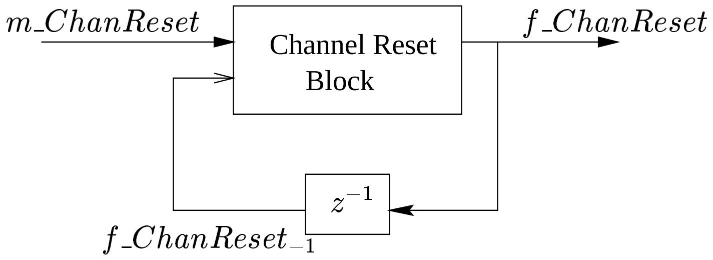


Figure 5.1: Block diagram for the channel reset

takes two input variables , *m_ChanReset* and $f\_ChanReset_{-1}$ and produces a single output $f\_ChanReset$. Here *m_ChanReset* denotes the reset switch status which has binary values, *on* or *off*. The output $f\_ChanReset$ indicates the current channel reset status. The symbol z^{-1} represents the system transform which produces a feedback input $f\_ChanRest_{-1}$.

In PVS, we use two enumerate types *t_Key* and *t_Reset* to define the types of the switch and of the channel reset state.

$$\texttt{t\_Key} : \texttt{TYPE} = \{\texttt{e\_KeyOn}, \texttt{e\_NotKeyOn}\}$$

$$\texttt{t\_Reset} : \texttt{TYPE} = \{\texttt{e\_WaitForOn}, \texttt{e\_WaitForOff}, \texttt{e\_ResetStuck}, \texttt{e\_ClearTrip}\}.$$

Here single letter prefixes or suffixes used in the PVS files denote a type (t), a function (f), a state variable (s), a monitor variable (m), a constant (k) or an enumerated value (e). Usually, we attach e and t to the beginning of the SRS variables or types and to the end of the SDD variables or types. For instance, the channel reset state *wait_for_on* is represented in the SRS as *e_waitForOn* while in the SDD it is $waitForOn\_e$.

5.2 Original Definitions

5.2.1 SRS Definition

The original Software Requirement Specification (SRS) of the channel reset block is shown in Table 5.1.

Table 5.1: Original SRS definition of the channel reset block

Condition Statement	1	2*	3	4	5	6*	7	8	9	10	11	12	13	14
(m_ChanReset = e_KeyOn) HELD_FOR (k_Stuck)	T	F	F	F	F	F	F	F	F	F	F	F	F	F
(m_ChanReset = e_KeyOn) HELD_FOR (k_Debounce)	-	T	T	T	T	T	F	F	F	F	F	F	F	F
(m_ChanReset = e_NotKeyOn) HELD_FOR (k_Debounce)	-	T	F	F	F	F	T	T	T	T	F	F	F	F
f_ChanReset_si	-	-	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr
Action Statements														
f_ChanReset = e_WaitForOn							X		X	X	X			X
f_ChanReset = e_WaitForOff			X	X								X		
f_ChanReset = e_ResetStuck	X				X								X	
f_ChanReset = e_ClearTrip								X						

Table 5.1 is a structured decision table whose format has been described in Section 2.4.2. In this table the *Held_For* operator is used to determine how long the reset switch has been activated or deactivated. Here we have two different *durations* used with the Held_For operator, i.e., the switch debounce time (*k_Debounce*) and the time after which the switch is considered stuck (*k_Stuck*). In our case, *k_Debounce* is at least 2 consecutive samples but less than 4 consecutive samples, and *k_Stuck* is at least 4 consecutive samples. The sampling period or loop time is 50 ms.

In Table 5.1 $f\_ChanReset\_si$ is the previous value of the channel reset state. The abbreviations, *won*, $woff$, *stk* and *clr*, correspond the channel reset state values as shown in 5.2.

Table 5.2: Abbreviations for the channel reset state values

state variable	*abbreviation*
wait_for_on	won
wait_for_off	woff
stuck	stk
cleartrip	clr

Using the above information, we can represent Table 5.1 in PVS as following:

```
% The type declaration of the SRS input
t_SrsInput: TYPE = [clock -> t_Key]

% The SRS function for non-inital state
f_ChanReset
 (m_ChanReset:t_SrsInput, f_ChanReset_si:t_Reset, t1:noninit_elem): t_Reset =
  IF  Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Stuck)(t1)
  THEN  e_ResetStuck
  ELSE
     COND
      Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
      NOT Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
       COND
             e_WaitForOn?(f_ChanReset_si)  ->  e_WaitForOff,
             e_WaitForOff?(f_ChanReset_si) ->  e_WaitForOff,
             e_ResetStuck?(f_ChanReset_si) ->  e_ResetStuck
       ENDCOND,

     NOT Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
        Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
           COND
              e_WaitForOn? (f_ChanReset_si) ->  e_WaitForOn,
              e_WaitForOff?(f_ChanReset_si) ->  e_ClearTrip,
              e_ResetStuck?(f_ChanReset_si) ->  e_WaitForOn,
              e_ClearTrip? (f_ChanReset_si) ->  e_WaitForOn
           ENDCOND,

     NOT Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
```

```
      NOT Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
            COND
               e_WaitForOn? (f_ChanReset_si) ->  e_WaitForOn,
               e_WaitForOff?(f_ChanReset_si) ->  e_WaitForOff,
               e_ResetStuck?(f_ChanReset_si) ->  e_ResetStuck,
               e_ClearTrip? (f_ChanReset_si) ->  e_WaitForOn
            ENDCOND
     ENDCOND
  ENDIF
```

The PVS function `f_ChanReset` takes three arguments, among which *m_ChanReset* represents the sequence of input values for the reset switch. It is of type *t_SrsInput*, whose domain are clock values and range are switch values of type *t_Key*. The other argument $t1$ is a non-initial clock value so the function `f_ChanReset` defines the state transition beyond the initial state. According to the specification, the initial channel trip reset state is $e\_WaitForOn$. Taking the initial value into consideration, we obtain the complete SRS specification in PVS through the addition of the following:

```
srs(m_ChanReset)(t): RECURSIVE t_Reset =
  IF init(t) THEN e_WaitForOn
  ELSE f_ChanReset(m_ChanReset,srs(m_ChanReset)(pre(t)),t)
  ENDIF
  MEASURE rank(t)
```

5.2.2 SDD Implementation

In the proposed implementation, a countdown timer is used to implement the Held_For operators in the SRS. The value of the countdown timer, *tim_c_key_s*, is defined in Table 5.3.

Table 5.3: Definition of the countdown timer in the SDD

Condition Statements	1	2
i_dip = keyOn_e	F	T
Action Statements		
tim_c_key_s = tim_stuck_c	X	
tim_c_key_s = MAX (0, tim_c_key_s -4)		X

This simple structure decision table shows that if the input reset switch (*i_dip*) is *on* (*keyOn_e*), the countdown timer values will decrease by 4 ms for every sample period until the counter reaches 0. But if the reset switch is *off*, the timer will be reset to its maximum value *tim_stuck_c*.

Considering that the switch stuck time (*k_Stuck*) is at least 4 sample instances, we can compute *tim_stuck_c* through the following equation:

$$k\_Stuck = CEILING((tim\_stuck\_c - 1)/4) \tag{5.1}$$

Here the CEILING function is used because we ignore the inter-sample clock value. For instance, CEILING (4.4) will round up to 5. In our case, $tim\_stuck\_c = 17$ since $k\_Stuck = CEILING(17 - 1)/4 = 4$. Figure 5.2.2 shows the relationship about the switch status ($KeySwitch$), the counter value ($tim\_c\_key\_s$) and $keyon\_Stuck$.

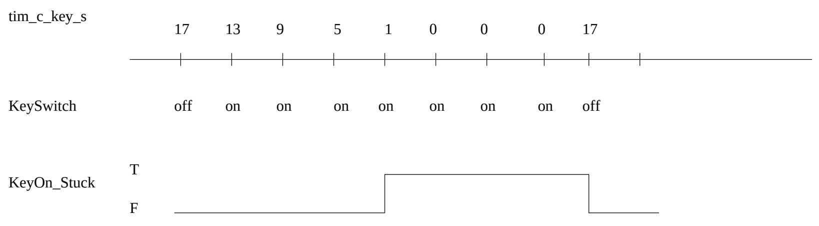


Figure 5.2: Diagram about the reset switch, *tim_c_key_s* and *keyon_Stuck*

As we see, when $tim\_c\_key\_s \leq 1$, the reset switch has been *on* for *k_Stuck* time and then the boolean variable *keyon_Stuck* is set to *true*.

The PVS version of Table 5.3 is:

```
gbl_word_t: TYPE = subrange(0, 65535)
dip_key_t:  TYPE = {keyon_e, keyoff_e}

tim_c_key_s(x: dip_key_t, tim_c_key_s_si: gbl_word_t): gbl_word_t =
  COND
    (x = keyon_e) -> max(0, tim_c_key_s_si-4)
    ELSE          -> tim_stuck_c
  ENDCOND
```

Note that *dip_key_t* is the SDD type declaration of the reset switch corresponding to *t_Key* in the SRS. This function returns a value of type *gbl_word_t*, a subrange of integers between 0 and 65535.

The complete SDD implementation is defined in Table 5.4, which consists of 3 structure decision tables. The first table is the major one, showing 4 input variables, i.e., *tim_c_key*, *i_dip*, *crk_pre_key_s* and *crk_chan_reset_si*, and one output *crk_chan_reset_s*. The second and third tables are actually the sub-tables of the first one, used to define the variables *crk_pre_key_s* and *tim_c_key* respectively.

Table 5.4: SDD definition of the channel reset block

Condition Statement	1	2	3	4	5	6*	7	8	9	10	11	12	13*	14	15	16	17
tim_c_key$\leq$ 1	T	F	F	F	F	F	F	F	F	F	F	F	F	-	-	-	-
i_dip = keyOn_e	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
crk_pre_key_s = keyon_e	-	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
crk_chan_reset_si	-	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr
Action Statement																	
crk_chan_reset_s = wait_for_on_e						X			X	X			X	X		X	X
crk_chan_reset_s = wait_for_off_e		X	X		X		X				X						
crk_chan_reset_s = reset_stuck_e	X			X				X				X					
crk_chan_reset_s = clear_trip_e															X		

Condition Statements	1	2
i_dip = keyOn_e	T	F
Action Statements		
crk_pre_key_s = keyon_e	X	
crk_pre_key_s = not_keyon_e		X

Condition Statements	1	2
i_dip = keyOn_e	F	T
Action Statements		
tim_c_key_s = tim_stuck_c	X	
tim_c_key_s = MAX (0, tim_c_key_s -4)		X

The PVS function `crk_chan_reset_s` below is based on Table 5.4.

```
crk_chan_reset_s(
   tim:   gbl_word_t,
dip_key:   dip_key_t,
crk_v: dip_key_t,
crk_chanReset_si: crk_reset_t): crk_reset_t = COND
   (tim <=1) & (dip_key = keyon_e)
         -> reset_stuck_e,
   NOT(tim <=1) & (dip_key = keyon_e) & (crk_v = keyon_e) ->
            COND
            (crk_chanReset_si = wait_for_on_e)  -> wait_for_off_e,
            (crk_chanReset_si = wait_for_off_e) -> wait_for_off_e,
            (crk_chanReset_si = reset_stuck_e)  -> reset_stuck_e,
            (crk_chanReset_si = clear_trip_e)  ->  wait_for_off,
            ENDCOND,
   NOT(tim <=1) & (dip_key = keyon_e) & NOT (crk_v = keyon_e) ->
            COND
            (crk_chanReset_si = wait_for_on_e)  -> wait_for_on_e,
            (crk_chanReset_si = wait_for_off_e) -> wait_for_off_e,
            (crk_chanReset_si = reset_stuck_e)  -> reset_stuck_e,
            (crk_chanReset_si = clear_trip_e)   -> wait_for_on_e
            ENDCOND,
   NOT(tim <=1) & NOT (dip_key = keyon_e) & (crk_v = keyon_e) ->
        COND
            (crk_chanReset_si = wait_for_on_e)  -> wait_for_on_e,
            (crk_chanReset_si = wait_for_off_e) -> wait_for_off_e,
            (crk_chanReset_si = reset_stuck_e)  -> reset_stuck_e,
            (crk_chanReset_si = clear_trip_e)   ->  wait_for_on_e
        ENDCOND,
   NOT(tim <=1) & NOT (dip_key = keyon_e) & NOT (crk_v = keyon_e) ->
        COND
            (crk_chanReset_si = wait_for_on_e)  -> wait_for_on_e,
            (crk_chanReset_si = wait_for_off_e) -> clear_trip_e,
            (crk_chanReset_si = reset_stuck_e)  -> wait_for_on_e,
            (crk_chanReset_si = clear_trip_e)   -> wait_for_on_e
        ENDCOND
   ENDCOND
```

Note that the function arguments *tim, dip_key, crk_v* and *crk_chanReset_si* correspond to the input variables of *tim_c_key, i_dip, crk_pre_key_s* and *crk_chan_reset_si* in Table 5.4.

Considering the initial conditions, the complete SDD implementation in PVS is:

```
sdd(i_dip)(t): RECURSIVE sdd_state_t =
  COND
  init(t) -> (#tim_c_key_s:=0, crk_chan_reset_s:=wait_for_on_e #),
  ELSE    -> (#tim_c_key_s:=
                f_tim_c_key(i_dip(t),
                  tim_c_key_s(sdd(i_dip)(pre(t)))),
                crk_chan_reset_s:= crk_chan_reset_s(
                  f_tim_c_key(i_dip(t), tim_c_key_s(sdd(i_dip)(pre(t)))),
                  i_dip(t),
                  i_dip(pre(t)),
                  crk_chan_reset_s(sdd(i_dip)(pre(t))))#)
  ENDCOND
  MEASURE rank(t)
```

The PVS function `sdd` returns a record construction *sdd_state_t*, which contains two fields, i.e., *tim_c_key_s* and *crk_chan_reset_s*. The PVS record construction and field selection have been described in Chapter 4. In this functions, we use the field selecting expression

`crk_chan_reset_s(sdd(i_dip)(pre(t))))`

to indicate that among the two returned values, only *crk_chan_reset_s* has been referenced.

5.3 Analysis of the Verification Results

Chapter 4 has described the two kinds of the proof obligation that must be attempted in the verification of the timing block. These are:

(1) The *Type Correctness Conditions (TCCs)* generated by PVS.

(2) The main block comparison theorem: $Abst_C \circ REQ = SOF_{req} \circ Abst_M$.

In this timing block, we interpret the proof obligation (2) as the following theorem

```
srs_eq_sdd : THEOREM
  Abst_Reset(srs(m_ChanReset)(t)) =
   crk_chan_reset_s(sdd(Abst_pre_keybit(m_ChanReset))(t))
```

where the abstraction functions *Abst_Reset* and *Abst_pre_keybit* correspond to $Abst_C$ and $Abst_M$. The function *srs* is *REQ* and the function *sdd* is SOF_{req}.

In this section, we first discuss the problem of the impossible state transition by exploring the generated TCCs. Later on, we will analyze the failed invariance proofs regarding theorem `srs_eq_sdd`, to diagnose the source of the failure in the original definitions and implementations.

5.3.1 Impossible state transitions

In Table 5.1 and Table 5.4, there are some columns marked with $*$, indicating those columns represent impossible state transitions. In other words, those states cannot be reached from the initial state. The original tables adopt different approaches to present those impossible state transitions. These approaches are:

(a) Assign an arbitrary state value to the result, as in Column 13 of Table 5.4,

(b) Leave a blank space in the result, as in Columns 2 and 6 of Table 5.1,

(c) Delete the related column from the table.

As mentioned earlier, PVS type-checking ensures the completeness and consistency of each TABLE or COND expressions. If we ignore the unreachable states in the PVS file as described in (b) and (c), the coverage TCCs that are generated will be unprovable.

The sequents discussed below are the subgoals of the TCCs which cannot be proved through the PVS automatic TCC proof strategy or manual input of the proof rules.

(1) In the SRS definition of Table 5.1, Column 6 does not show the transition result in the action grid. Hence we omit that transition when *e_ResetStuck?(f_ChanReset_si)* is *true* in the following PVS file:

```
Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
NOT Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
 COND
      e_WaitForOn?(f_ChanReset_si)  ->  e_WaitForOff,
                e_WaitForOff?(f_ChanReset_si) ->  e_WaitForOff,
                e_ResetStuck?(f_ChanReset_si) ->  e_ResetStuck
          ENDCOND,
```

Since *f_ChanReset_si* is of an enumerate type, including four possible state values, i.e., *e_WaitForOn, e_WaitForOff, e_ResetStuck* and *e_ResetStuck.* Missing one of the four cases will results in a coverage TCC.

```
f_ChanReset_TCC4.1.1.1 :

[-1]    e_ClearTrip?(f_ChanReset_si!1)
[-2]    e_KeyOn?(m_ChanReset!1(50))
[-3]    e_KeyOn?(m_ChanReset!1(0))
  |-------
```

Since we lack sufficient information about the relationship between *f_ChanReset_si!1* and *m_ChanReset!1* at current step, this sequent becomes unprovable.

(2) The following is another unprovable sequent regarding the coverage TCC of *crk_chan_reset_s_TCC10.*

```
crk_chan_reset_s_TCC10 :
{-1}    (tim!1 <= 1)
  |-------
{1}    keyon_e?(dip_key!1)
```

In the original implementation of Table 5.4, we define the transition under the input combination of

$$(tim!1 \leq 1) \wedge (dip\_key!1 = keyon\_e)$$

but ignore the counter-input combination such as

$$(tim!1 \leq 1) \wedge (dip\_key!1 \neq keyon\_e).$$

So the above sequent `crk_chan_reset_s_TCC10` requires us to prove that

$$\neg((tim!1 \leq 1) \wedge (dip\_key!1 \neq keyon\_e))$$

Based on the current information, we cannot prove this sequent.

Although the above TCCs cannot be proved, the original specification and implementation are still fine because these states cannot be reached from the initial state. We confirm this observation by performing “refutation” in PVS. The following theorems refute the existence of an input combination leading to these impossible transition. These theorem have been proved successfully.

```
impossible_srs_c6: theorem FORALL (t: noninit_elem):
 NOT (Held_For(LAMBDA (t:noninit_elem):
       e_KeyOn?(m_ChanReset(t)), k_Debounce)(t)
      AND e_ClearTrip?(srs(m_ChanReset)(pre(t))))

impossible_sdd_c1_1: theorem FORALL (t: noninit_elem):
  NOT (keyoff_e?(Abst_pre_key(m_ChanReset)(t)) &
         (tim_c_key_s(Abst_Key(m_ChanReset(t)),
             tim_c_key_s(sdd(m_ChanReset)(pre(t)))) <= 1))
```

As mentioned at the beginning of this subsection, the original tables try 3 different ways to represent the impossible transitions. Only the first approach can avoid the unprovable TCCs, that is, each unreachable state is assigned a result to avoid the coverage TCCs. We will employ this approach when we provide the “fixed” versions of the SRS and SDD in Section 5.4.

5.3.2 Implicit assumption on the initial input

Once we have eliminate all of the unprovable TCCs of the SRS and SDD definitions, we can attempt to prove the main block comparison theorem `srs_eq_sdd`. We first specify and prove some of the important lemmas which relate the *SRS condition* to the *SDD condition.*

The following lemma `srs_sdd_keyon_heldfor_stuck` shows that, for any non-initial clock value of t, if

$$Held\_For(LAMBDAt : e\_KeyOn?(m\_ChanReset(t)), k\_Stuck)(t)$$

is *true* in the SRS, then

$$tim\_c\_key\_s \leq 1ANDAbst\_Key(m\_ChanReset(t)) = keyon\_e$$

is *true* in the SDD, and vice versa. In other words, we want to show that Column 1 in Table 5.1 corresponds to Column 1 in Table 5.4.

```
srs_sdd_keyon_heldfor_stuck: lemma
  FORALL (t: noninit_elem):
   Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Stuck)(t)
         IFF
   (tim_c_key_s(Abst_Key(m_ChanReset(t)),
          tim_c_key_s(sdd(m_ChanReset)(pre(t)))) <= 1)
    & keyon_e?(Abst_Key(m_ChanReset(t)))
```

Attempting to prove this lemma results in the following unprovable sequent:

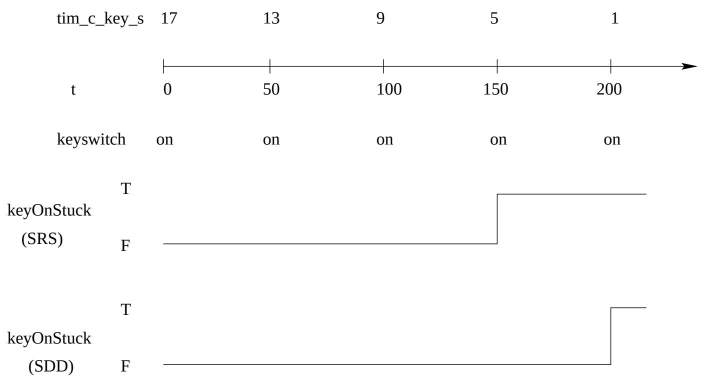
```
srs_sdd_keyon_heldfor_stuck.1.1.2.2.2.1.1 :

[-1]    e_KeyOn?(m_ChanReset!1(150))
[-2]    e_KeyOn?(m_ChanReset!1(100))
[-3]    e_KeyOn?(m_ChanReset!1(50))
[-4]    e_KeyOn?(m_ChanReset!1(0))
  |-------
```

This unprovable sequent requires us to prove

```
 NOT  (e_KeyOn?(m_ChanReset!1(150)) AND
      e_KeyOn?(m_ChanReset!1(100)) AND
      e_KeyOn?(m_ChanReset!1(50))  AND
      e_KeyOn?(m_ChanReset!1(0)) )
```

In other words, the sequent demands that the switch cannot be set to *On* for 4 consecutive sample instances starting from the initial state. Otherwise, *keyOn_stuck* is *true* according to the SRS, whereas it is *false* according to the SDD. The following diagram shows the behavior of the SRS and SDD for the problematic sequent.



This problem results from the implicit (or undocumented) assumption on the initial value of $m\_ChanReset!1(t)$. This original specification assigns *not_keyon_e* to the initial value of *crk_pre_key_s* but does not document the initial value of $m\_ChanReset(t)$. If $m\_ChanReset(0)$ is initialized to e_KeyOff, then line [-4] in the above sequent is *false* and the lemma `srs_sdd_keyon_heldfor_stuck` will be proved instantly.

The assumption can be made explicit by using a subtype of `t_SrsInput` to place the restriction on $m\_ChanReset!1$.

$$\texttt{m\_ChanReset} : \texttt{VAR}\{\texttt{m} : \texttt{t\_SrsInput} | \texttt{m}(0) = \texttt{e\_NotKeyOn}\}$$

With this restriction, the lemma `srs_sdd_keyon_heldfor_stuck` is now easily proved.

5.3.3 Wrong initial value for the countdown timer

As mentioned earlier, *tim_c_key_s* is a countdown timer for the reset switch. Its initial value is assigned to 0.

Before we prove the equivalence of the SRS and SDD for arbitrary value of t, we consider some special values of t at first. This idea is another interesting feature of the *PVS-RT* methodology: If the verifier believes that the SRS and SDD are not equivalent for a particular input combination, he can try to prove a theorem to that effect to confirm his intuition.

Let $t = 50$, we rewrite the equivalence theorem `srs_eq_sdd` by substituting the value 50 for t to obtain:

```
srs_eq_sdd_t50 : THEOREM
  Abst_reset_s(srs(m_ChanReset)(50)) =
  crk_chan_reset_s(sdd(m_ChanReset)(50))
```

Attempting the proof of this theorem results in two unprovable sequents, one of which is

```
srs_eq_sdd_t50:
{-1}    e_KeyOn?(m_ChanReset!1(50))
  |-------
[1]
```

This sequent ask us to prove that

$$NOT \;\; e\_KeyOn?(m\_ChanReset!1(50))$$

Unfortunately, we cannot guarantee that $m\_ChanReset!1(50)$ is not e_KeyOn because it is a monitored variable. This unprovable sequent force us to re-consider the original definitions and the initial settings.

As shown in Figure 5.3, suppose the switch is set to *off, on, on, on* and *on* while t is 0, 50, 100, 150, respectively. When $t = 50$, the previous value of *tim_c_key_s* is 0 and at the current state, $m\_ChanReset!1(50) = e\_KeyOn$. According to the following function:

```
tim_c_key_s(x: dip_key_t,
   tim_c_key_s_si: gbl_word_t): gbl_word_t =
      COND
      (x = keyon_e)
          -> max(0, tim_c_key_s_si-4),
      ELSE
          -> tim_stuck_c
      ENDCOND
```

the current value of *tim_c_key_s* will remain 0 because $max(0, 0 - 4) = 0$. Under this condition, Column 1 in Table 5.4 is satisfied so *keyOn_Stuck* is true. Since the switch stuck time *k_Stuck* is at least 4 sample instance, the behavior of the SDD is clearly incorrect.

If we change the initial value of *tim_c_key_s* to the constant *tim_stuck_c* instead of 0, we can solve this problem. The basic idea is: when $t = 50$ and $m\_ChanReset!1(50) = e\_KeyOn$, for the SRS, none of the the Held_For operators is *true*, so Column 11 in Table 5.1 is satisfied and the next transition state is $e\_WaitForOn$; for the SDD, $tim\_c\_key\_s = max(0, 17 - 4) = 13$, Column 6 in Table 5.4 is satisfied so the next transition state is $wait\_for\_on\_e$. Therefore, the SDD implements the SRS correctly.

5.4 The "Fixed" Definitions

Taking the above information into consideration, we provide the "fixed" versions of the specification and implementation in Table 5.5 and Table 5.6, where all of the TCCs

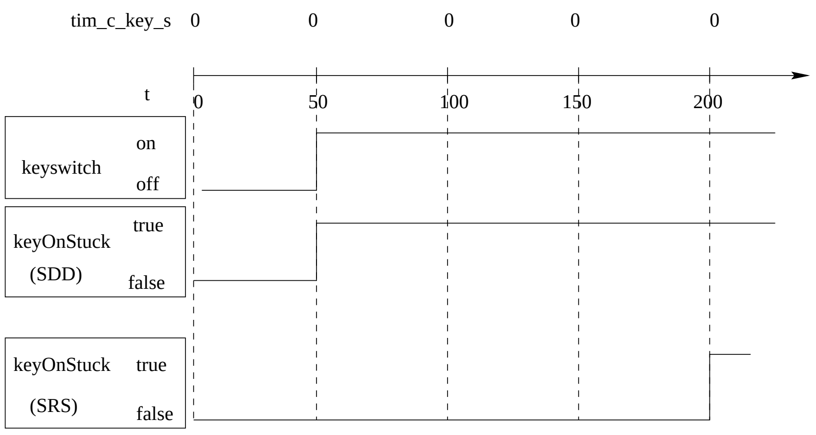


Figure 5.3: The problem of the initialization about the countdown timer

have been proved. The initial values of the *m_ChanReset* and *tim_c_key_s* have been re-defined in the complete PVS specification file. The interested reader is referred to Appendix F.1 for the details.

Table 5.5: "Fixed" SRS definition of channel reset

Condition Statement	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(m_ChanReset = e_KeyOn) HELD_FOR (k_Stuck)	T	F	F	F	F	F	F	F	F	F	F	F	F	F
(m_ChanReset = e_KeyOn) HELD_FOR (k_Debounce)	-	T	T	T	T	T	F	F	F	F	F	F	F	F
(m_ChanReset = e_NotKeyOn) HELD_FOR (k_Debounce)	-	T	F	F	F	F	T	T	T	T	F	F	F	F
f_ChanReset_si	-	-	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr
Action Statements														
f_ChanReset = e_WaitForOn							X		X	X	X			X
f_ChanReset = e_WaitForOff		X	X	X		X						X		
f_ChanReset = e_ResetStuck	X				X								X	
f_ChanReset = e_ClearTrip								X						

Table 5.6: “Fixed” SDD implementation of the channel reset block

Condition Statement	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
tim$c_key≤ 1	T	T	F	F	F	F	F	F	F	F	F	F	F	F	-	-	-	-
i_dip = keyOn_e	T	F	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
crk$pre_key_s = keyon_e	-	-	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
crk$chan_reset_s	-	-	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr	won	woff	stk	clr
Action Statement																		
crk$chan_reset_s = wait_for_on_e							X			X	X			X	X		X	X
crk$chan_reset_s = wait_for_off_e			X	X		X		X				X						
crk$chan_reset_s = reset_stuck_e	X	X			X				X				X					
crk$chan_reset_s = clear_trip_e																X		

5.5 The Proof Strategies

Before the actual proof is carried out, we specify and prove some useful lemmas which relate the *SRS conditions* to the *SDD conditions*. For instance, if the reset switch has been held *on* for a debounce time, the *SRS condition* is represented as

```
 Held_For (LAMBDA (t:noninit_elem):e_KeyOn?(m_ChanReset(t)), k_Debounce)(t)
```

and the *SDD condition* is shown as

```
 keyon_e?(Abst_pre_key(m_ChanReset)(t)) &
 keyon_e?(Abst_pre_key(m_ChanReset)(pre(t)))
```

Then the following lemma intends to show that the above *SRS condition* corresponds to the *SDD condition*.

```
srs_sdd_keyon_heldfor_debounce: lemma  FORALL (t: noninit_elem):
 Held_For
  (LAMBDA (t:noninit_elem):e_KeyOn?(m_ChanReset(t)), k_Debounce)(t)
 iff  keyon_e?(Abst_pre_key(m_ChanReset)(t)) &
 keyon_e?(Abst_pre_key(m_ChanReset)(pre(t)))
```

The following are two other similar lemmas regarding the reset switch having been *on* for stuck time or been *off* for debounce time. All of these lemmas have been proved before been used in the proof of the main block comparison theorem. The proof scripts are listed in Appendix F.2.

```
srs_sdd_keyon_heldfor_stuck: lemma
 FORALL (t: noninit_elem):
 Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Stuck)(t)
 IFF  (tim_c_key_s(Abst_Key(m_ChanReset(t)),
 tim_c_key_s(sdd(m_ChanReset)(pre(t)))) <= 1)

srs_sdd_NotKeyon_heldfor_debounce: lemma
FORALL (t: noninit_elem):
 Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t)
 IFF
 keyoff_e?(Abst_Key(m_ChanReset(t))) &
    keyoff_e?(Abst_Key(m_ChanReset(pre(t))))
```

In order to prove the main block comparison theorem `srs_eq_sdd`, we use the strategy `(INDUCT ‘‘t’’ 1 ‘‘clock_induction)`. This break the proof into two parts: (i) Base case when $t = 0$; (ii) inductive case. The theorem can be easily proved in the base case since the initial values of the state variables are strictly defined.

For the inductive step, we assume the SRS corresponds to the SDD in the previous state ($pre(t)$). If we know that SRS(pre(t)) implies SRS(t) and thus can prove SDD(pre(t)) implies SDD(t), we can conclude that the SDD corresponds to SRS for any clock value t.

Knowing that SRS(pre(t)) implies SRS(t), we can figure out under which *SRS conditions* this transition (i.e., SRS(pre(t)) implies SRS(t)) is accomplished. Bearing the *SRS conditions* in mind, we can use the *equivalence lemmas* to find the corresponding *SDD conditions*. Given the SDD(pre(t)) and the *SDD conditions*, we can compute *SDD(t)*.

We illustrate this proof strategy through the following example:

```
[-1] srs(m_ChanReset!1)(t!1) = e_ResetStuck
[-2] srs(m_ChanReset!1)(pre(t!1)) = e_WaitForOff
[-3] (wait_for_off_e = crk_chan_reset_s(sdd(m_ChanReset!1)
            (pre(t!1))))
  |-------
[1]  (reset_stuck_e = crk_chan_reset_s(sdd(m_ChanReset!1)(t!1)))
```

In this sequent, assume both SRS and SDD are in the state of $wait\_for\_off$ at $pre(t)$ (see [-2] and [-3]) and SRS in current state is *ResetStuck* ([-1]), we have to prove that the SDD in current state is *ResetStuck* as well ([1]).

Step 1:

Using PVS commands (EXPAND "srs" -1) (EXPAND "f_ChanReset2") (LIFT-IF) (ASSERT) (BDDSIMP), we get the following sequents and the *SRS condition* is in line[-1]:

```
srs_eq_sdd.2.7 :

{-1} Held_For(LAMBDA (t: clock[50]): e_KeyOn?(m_ChanReset!1(t)),
       k_Stuck)(t!1)
{-2} e_WaitForOff?(srs(m_ChanReset!1)(pre(t!1)))
{-3} wait_for_off_e?(crk_chan_reset_s(sdd(m_ChanReset!1)(pre(t!1))))
  |-------
{1}  reset_stuck_e?(crk_chan_reset_s(sdd(m_ChanReset!1)(t!1)))
{2}  e_WaitForOff?(srs(m_ChanReset!1)(t!1))
{3}  e_WaitForOn?(srs(m_ChanReset!1)(t!1))
```

Step 2:

Then use (LEMMA "srs_sdd_keyon_heldfor_stuck") to find the corresponding *SDD conditions*, which are in line [-2] of the sequent below:

```
[-1] Held_For(LAMBDA (t: clock[50]): e_KeyOn?(m_ChanReset!1(t)),
     k_Stuck)(t!1)
{-2} (tim_c_key_s(Abst_Key(m_ChanReset!1(t!1)),
     tim_c_key_s(sdd(m_ChanReset!1)(pre(t!1)))) <= 1)
[-3] e_WaitForOff?(srs(m_ChanReset!1)(pre(t!1)))
[-4] wait_for_off_e?(crk_chan_reset_s(sdd(m_ChanReset!1)(pre(t!1))))
  |-------
[1]  reset_stuck_e?(crk_chan_reset_s(sdd(m_ChanReset!1)(t!1)))
[2]  e_WaitForOff?(srs(m_ChanReset!1)(t!1))
[3]  e_WaitForOn?(srs(m_ChanReset!1)(t!1))
```

Step 3:

After hiding the irrelevant formulas, we get the following sequents:

```
[-1]    (tim_c_key_s(Abst_Key(m_ChanReset!1(t!1)),
     tim_c_key_s(sdd(m_ChanReset!1)(pre(t!1)))) <= 1)
```

```
[-2]    wait_for_off_e?(crk_chan_reset_s(sdd(m_ChanReset!1)(pre(t!1))))
  |-------
[1]    reset_stuck_e?(crk_chan_reset_s(sdd(m_ChanReset!1)(t!1)))
```

This transition (i.e. $[-1]\&[-2] => [1]$) can be achieved easily because it was defined in Column 1 of the SDD table.

As shown in Figure 5.4, there are as much as 16 main subgoals encountered after applying propositional simplification (i.e., `bddsimp`) in the proof of the block comparison theorem. Each main subgoals may consist of some sub-subgoals and every sequent may contain several formulas. Experience shows that blindly applying the PVS commands such as (`expand`) and (`grind`) may make the proof too complicated to finish. The choice of this proof strategy is crucial in that it provides guidelines as to how to accomplish the theorem proof effectively and efficiently.

5.6 Summary

To verifying the channel trip reset block, we first define the original SRS specification and the SDD implementation in PVS. Our attempts to prove the main block comparison theorem and the TCCs failed and result in some unprovable sequents. Analyzing the unprovable sequents leads to find the errors, implicit assumption and undocumented impossible state transitions in the original definitions. After fixing those errors, we provided the updated version of the SRS and SDD and proved all of the TCCs and the main block comparison theorem. So we can conclude the modified SRS specification and SDD implementation have equivalent functionality for all the input values. The PVS proof strategies summarized in the later part provide guidance for the future verification.

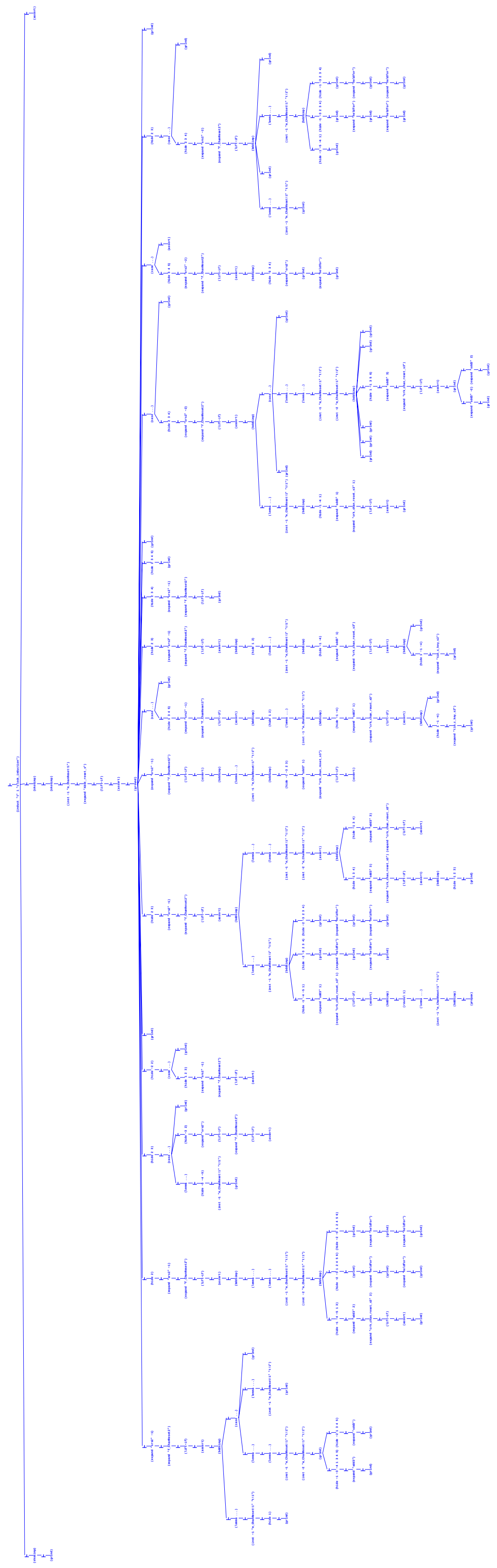


Figure 5.4: Proof tree structure of the main block comparison theorem

Chapter 6

Conclusions

This chapter summaries the results of this thesis, discusses its limitations and describes future work.

6.1 Results

The thesis refines the PVS Real-Time method (*PVS-RT*) outlined in [13], focusing on the development of the PVS library for the specification and verification of real-time control systems.

In [4], the general clock induction theorem is provided which can be used to employ the general mathematical induction over clock values. From this definition, we derive strong clock induction theorem so that one is allowed to used a broader induction hypothesis in the theorem proof.

The thesis also provides a detailed description of the formal verification of two timing blocks of an industrial real-time control system. The PVS specification and proof techniques are described in sufficient details to show how the errors or invalid assumptions are detected in the original specification and the proposed implementation. The "fixed" versions of the SRS and SDD definitions are interactively developed using PVS. More specifically, PVS is used to debug new implementations, and verify that the final implementation corrected the detected errors and was equivalent to the specification for all possible input trajectories. PVS also helped debug the SRS specification to make a rigorous and "safer" requirement.

The main benefit of the *PVS-RT* method is that it delivers a guarantee of domain coverage. It checks all possible input sequences, and in the case when the SRS and SDD are not equivalent it provides some insight into the reasons for any discrepancies. Moreover, when a verifier suspects discrepancy, he can conduct “refutation” theorem proving to confirm that the implementation does not satisfy the specification. Properly applied this method for the verification of timing blocks provides an increased level of confidence in the verification process and aids in detecting subtle timing errors.

6.2 Limitations and Future Research

The work currently has several limitations:

(1) The current implementation assumes that both the SRS and SDD operate at the same sampling rates. For mulitple sampling rates that are related by relational mulitples, we could always choose our sample rate based upon the least common multiple. This would add some complexity to the function descriptions and result in a larger state space, especially for the recursive definition.

(2) The intersample behavior of the inputs is ignored. All the above mentioned verification methods requires additional reasoning regarding input filtering;

(3) Timing tolerances in the system specifications are not taken into consideration. The rigorous argument step dealing with timing tolerances may be required but could quite possibly be posed as an additional PVS proof obligation through an extension of [12] to timing properties.

(4) The verification of two timing blocks in Chapter 4 and 5 uses reduced timing delays in order to simplify the proofs involved “spooling” the recursive definition of the Held_For operator. Attempting proof by induction for actual timing values should be considered in the future.

Therefore the following areas are suggested for future work:

- Consideration of different clock rates used in the SRS and SDD theoretical models;

- Consideration of intersample behavior used in the SRS and SDD theoretical models;
- Verification of properties involving timing tolerances;
- Consideration of actual timing values in the block comparison;

6.3 Conclusions

The goal of this thesis is to provide sufficient knowledge as to how the PVS Real-Time method (*PVS-RT*) can be used to verify simple timing properties. The verification results have confirmed the potential benefits of formal requirement analysis with the help of automated theorem provers such as PVS. The general consensus is that the *PVS-RT* could be used at least as a secondary methods of verification and with some more work could reasonably be expected to serve as the primary method of verification for simple timing properties.

Bibliography

Appendix A

PVS Theories

In this appendix we provide the complete PVS specification files for the `Clocks`, `Held_For`, `timing_lemmas`, `c_sealedin` and `ChanReset` theories. For the limited space of the thesis, we do not list all of the proof files here, but rather present some important ones mentioned in the previous chapters. The interested reader may contact the author for the complete proof files.

A PVS for Clocks Theory

This section presents the PVS specification and proofs for the `Clocks` theory. This theory is originally defined by Dutertre and Stavridou in [4]. We enrich this theory by introducing the strong clock induction theorem.

A.1 PVS specification for the `Clocks` theory

Below is the specification of the discrete `Clocks` theory, including the basic type declaration, operator definitions and clock induction theorems.

```
Clocks[ K: posreal ]: THEORY
BEGIN
non_neg: TYPE = { x: real | x>=0 } CONTAINING 0
time: TYPE = non_neg
t: VAR time
n: VAR nat
```

```
clock: TYPE = { t: time | EXISTS(n: nat): t=n*K } CONTAINING 0
x: VAR clock
init(x): bool = (x=0)
noninit_elem: TYPE ={ x | not init(x) }
y: VAR noninit_elem
pre(y): clock = y - K
next(x): noninit_elem = x + K
rank(x): nat = x/K
clock_induct : LEMMA
FORALL (P : pred[clock]) :
    (FORALL x, n : rank(x) = n implies P(x))
implies
    (FORALL x : P(x))

% The general clock induction proposition
clock_induction: PROPOSITION
  FORALL (P: pred[clock]):
    (FORALL (t: clock): init(t)
      IMPLIES P(t)) AND
    (FORALL (t: noninit_elem): P(pre(t))
      IMPLIES P(t))
      IMPLIES (FORALL (t: clock): P(t))

% The strong clock induction proposion
clock_strong_induction: PROPOSITION
  FORALL (P: pred[clock]):
    (FORALL (x:clock):
    (FORALL (z:clock): z < x IMPLIES P(z)) IMPLIES P(x))
      IMPLIES (FORALL (t: clock): P(t))
END Clocks
```

A.2 PVS proofs of the `Clocks` theory

In this subsection we present 3 theorem proof files with respect to the clock induction. The lemma `clock_induct` is used to prove the general clock induction theorem `clock_induction`, whereas the strong clock induction theorem `clock_strong_induction` is proved through the use of the general clock induction theorem. Below are the associated proof scripts:

The `clock_induct` lemma:

```
(""
  (SKOSIMP)
  (SKOLEM!)
  (INST -1 "x!1" "rank(x!1)")
  (ASSERT))
```

The `clock_induction` proposition:

```
(""
 (SKOSIMP)
 (LEMMA "clock_induct" ("P" "P!1"))
 (SPLIT)
 (("1" (PROPAX))
  ("2"
   (DELETE 2)
   (INDUCT "n")
   (("1"
     (SKOSIMP)
     (INST -2 "x!1")
     (GRIND)
     (LEMMA "div_eq_zero" ("x" "x!1" "n0z" "K"))
     (GRIND))
    ("2"
     (SKOSIMP*)
     (ASSERT)
     (GROUND)
     (INST -1 "pre(x!1)")
     (("1" (INST -4 "x!1") (("1" (GRIND)) ("2" (GRIND)))) ("2" (GRIND))))))))
```

The `clock_strong_induction` proposition:

```
(""
 (LEMMA "clock_induction")
 (SKOSIMP*)
 (INST -1 "(LAMBDA (n:clock): (FORALL (m:clock): m <= n IMPLIES P!1(m)))")
 (BETA)
 (SPLIT)
 (("1" (INST -1 "t!1") (INST -1 "t!1") (ASSERT))
  ("2" (SKOSIMP*) (INST -3 0) (("1" (GRIND)) ("2" (GRIND))))
  ("3"
   (SKOSIMP*)
   (INST -3 "m!1")
   (ASSERT)
   (SKOSIMP)
   (INST -1 "z!1")
   (GRIND)
   (HIDE 2 3 4)
   (TYPEPRED "m!1" "z!1" "t!2" "K")
   (BOTH-SIDES "/" "K" -7)
   (("1"
     (BOTH-SIDES "/" "K" -9)
     (BOTH-SIDES "/" "K" -10)
     (BOTH-SIDES "/" "K" 2)
     (GRIND))
    ("2" (GRIND))))))
```

B PVS for Held_For Operator

This section presents the PVS specification and proofs for the Held_For theory. This theory is originally defined in [13]. We enrich this theory by introducing an alternative, ”set” definition of the Held_For operator.

B.1 PVS specification for Held_For theory

The PVS Held_For theory uses the Clocks theory defined before. So we import the Clocks theory at the beginning. The complete theory includes the recursive (i.e., heldfor and Held_For) and the ”set” definitions (i.e., Held_For_set) of the Held_For operator and the equivalent theorems with respect to the two definitions.

```
Held_For  [K:posreal] : THEORY

  BEGIN
  IMPORTING Clocks[K]
  t, t_now, t1: VAR clock
  duration:VAR time
```

```
 timed_cond:TYPE = [clock -> bool]
 P, tcond:var timed_cond

 % the recursive definition of the Held_For operator
 heldfor(P, t, t_now, duration):
   RECURSIVE bool =
       IF P(t) THEN
         IF (t_now - t >= duration) THEN TRUE
         ELSIF init(t) THEN FALSE
         ELSE heldfor(P,pre(t),t_now,duration)
         ENDIF
       ELSE FALSE
       ENDIF
       MEASURE rank(t)

  Held_For(P, duration): pred[clock] =
   (LAMBDA (t:clock): heldfor(P,t,t,duration))

 % the "set" definition of the Held_For operator
 Held_For_set (tcond, duration): timed_cond =
    (lambda (t_now):
       Exists (t): (t_now - t >=duration) and
        forall (t_i:{t_k: clock|(t<=t_k) & (t_k<=t_now)}):
            tcond (t_i))

 % Equivalence proof of the two definitions
 set_implies_recursive: theorem
   Held_For_set(tcond, duration)(t) IMPLIES  Held_For (tcond, duration)(t)

 recursive_implies_set: theorem
   Held_For(tcond, duration)(t) IMPLIES  Held_For_set (tcond, duration)(t)

 equivalence_Heldfor:  theorem
   Held_For(tcond, duration)(t) =  Held_For_set (tcond, duration)(t)

END Held_For
```

B.2 PVS proofs of the `Held_For` theory

In this subsection we present the proof scripts with respect to the equivalence of the two Held_For definitions. The theorem `set_implies_recursive` is proved successfully, showing that the "set" version implies that recursive one. The theorem `equivalence_Heldfor` has not been finished yet. We will discuss the encountered unproved sequents short after the proof script.

Below are the associated proof scripts:

The theorem `set_implies_recursive`:

```
(""
 (SKOSIMP)
 (EXPAND "Held_For_set")
 (EXPAND "Held_For")
 (SKOSIMP)
 (CASE "FORALL (t1 | t!2 <= t1 & t1 <= t!1): heldfor(tcond!1, t1, t!1, duration!1)")
 (("1" (HIDE -3) (INST?) (ASSERT))
  ("2"
   (HIDE 2)
   (INDUCT "t1" 1 "clock_induction")
   (("1" (ASSERT)) ("2" (ASSERT)) ("3" (GRIND))
    ("4"
     (SKOSIMP)
     (EXPAND "heldfor" +)
     (ASSERT)
     (PROP)
     (("1" (INST?))
      ("2"
       (EXPAND "pre" 3)
       (ASSERT)
       (CASE "t!2 < y!1")
       (("1"
         (LEMMA "lt_le_lemma")
         (INST -1 "t!2" "y!1")
         (HIDE --3 -4 -5 -6 -7 1 2)
         (HIDE -1)
         (HIDE -2)
         (GRIND)
```

```
        (TYPEPRED "t!2" "y!1")
        (GRIND)
        (BOTH-SIDES "/" "loop_time" -7)
        (BOTH-SIDES "/" "loop_time" 2)
        (GRIND))
       ("2" (GRIND))))
     ("3" (GRIND)) ("4" (GRIND)) ("5" (GRIND)))))))
```

The theorem `equivalence_Heldfor`**:**

```
 (""
(INDUCT "t" 1 "clock_induction")
(("1" (SKOSIMP*) (EXPAND* "Held_For" "Held_For_set") (EXPAND "heldfor") (GRIND))
 ("2"
  (SKOSIMP*)
  (INST -1 "duration!1" "tcond!1")
  (EXPAND "Held_For")
  (EXPAND "heldfor")
  (LIFT-IF)
  (ASSERT)
  (AUTO-REWRITE-THEORY "Held_For")
  (DO-REWRITE)
  (GROUND)
  (("1"
    (HIDE 2)
    (GROUND)
    (LIFT-IF)
    (ASSERT)
    (BDDSIMP)
    (("1"
      (EXPAND "Held_For_set")
      (INST 1 "pre(y!1)")
      (GROUND)
      (SKOLEM-TYPEPRED)
      (LEMMA "eq_innner_clcokvar")
      (INST -1 "y!1" "t_i!1")
      (GRIND))
     ("2" (GRIND))))
   ("2" (GRIND))
```

```
("3"
 (GROUND)
 (LIFT-IF)
 (ASSERT)
 (BDDSIMP)
 (("1"
   (LEMMA "Held_For_set_next")
   (INST -1 "duration!1" "tcond!1" "y!1")
   (GRIND))
  ("2"
   (HIDE 2 3 4)
   (EXPAND "Held_For_set")
   (INST 1 "pre(y!1)")
   (GROUND)
   (SKOLEM-TYPEPRED)
   (LEMMA "eq_innner_clcokvar")
   (INST -1 "y!1" "t_i!1")
   (GRIND))
  ("3"
   (HIDE -2 -4 1)
   (LEMMA "Held_For_next")
   (INST -1 "duration!1" "tcond!1" "y!1")
   (GRIND)
   (EXPAND "heldfor")
   (PROPAX))
  ("4"
   (EXPAND "heldfor")
   (ASSERT)
   (BDDSIMP)
   (("1" (GRIND)) ("2" (HIDE 6 7 5) (EXPAND "pre") (POSTPONE))
    ("3"
     (HIDE 4 3)
     (EXPAND "Held_For_set")
     (SKOLEM-TYPEPRED)
     (GROUND)
     (INST -5 "pre(pre(y!1))")
     (LEMMA "eq_inner3_clockvar")
     (INST -1 "y!1" "t!1")
     (GRIND))))
```

```
      ("5"
       (LEMMA "Held_For_set_next")
       (INST -1 "duration!1" "tcond!1" "y!1")
       (GRIND))
      ("6"
       (EXPAND "heldfor")
       (ASSERT)
       (GROUND)
       (("1"
         (HIDE 1 3 4)
         (EXPAND "Held_For_set")
         (INST 1 "pre(pre(y!1))")
         (GROUND)
         (SKOLEM-TYPEPRED)
         (TYPEPRED "y!1")
         (LEMMA "eq_inner3_clockvar")
         (INST -1 "y!1" "t_i!1")
         (GRIND))
        ("2"
         (HIDE 1 2 3 5 6 -1)
         (EXPAND "Held_For_set")
         (INST 1 "pre(pre(y!1))")
         (GROUND)
         (SKOLEM-TYPEPRED)
         (LEMMA "eq_inner3_clockvar")
         (INST -1 "y!1" "t_i!1")
         (GRIND))
        ("3" (HIDE -1 1 4 7 8 2) (EXPAND "pre") (GROUND) (POSTPONE))))))
    ("4" (GRIND))
    ("5"
     (HIDE 3)
     (EXPAND "heldfor")
     (EXPAND "Held_For_set")
     (SKOSIMP)
     (INST -3 "pre(y!1)")
     (GROUND)
     (("1"
       (TYPEPRED "t!1" "y!1")
       (GRIND)
       (HIDE -7 3 1)
       (LEMMA "ge_nat_duration")
       (INST -1 "duration!1" "n!2" "n!1")
       (GRIND))
      ("2" (GRIND))))
    ("6" (GRIND))))))
```

Analysis of the unprovable sequents

After employing the above proof strategies for the theorem `equivalence_Heldfor`, there are two unprovable sequents shown as follows:

```
equivalence_Heldfor.2.3.6.3 :

[-1]    heldfor(tcond!1, y!1 - 3 * K, y!1, duration!1)
[-2]    tcond!1(y!1 - 2 * K)
[-3]    tcond!1(y!1)
[-4]    tcond!1(y!1 - K)
  |-------
[1]    (K >= duration!1)
[2]    (2 * K >= duration!1)
[3]    Held_For_set(tcond!1, duration!1)(y!1)

equivalence_Heldfor.2.3.4.2 :

[-1]    tcond!1(y!1)
[-2]    Held_For_set(tcond!1, duration!1)(y!1)
{-3}    tcond!1(y!1 - K)
  |-------
{1}    (K >= duration!1)
{2}    (2 * K >= duration!1)
{3}    heldfor(tcond!1, y!1 - 3 * K, y!1, duration!1)
{4}    init(y!1 - K)
```

We note that the expression $2 * K \geq duration!1$ appears in the consequents of both sequents. If we keep expanding the heldfor definition n times, we will encounter a similar expression like $(n * loop\_time >= duration!1)$. Since we do not know the concrete values of the *duration* and K, n is uncertain and the proof will be endless.

However, given concrete values of K and *duration*!, we can always find a natural number n such that $(n * loop\_time >= duration)$. We prove this observation by specifying and proving the following PVS lemma:

```
loop_duration_n: LEMMA (EXISTS n: nat): n*K >= duration
```

The interested reader is referred to Appendix C.2 for the detailed proof commands.

B.3 Comparison of the two definitions

The theory `simple` below contains two similar theorems written in different Held_For definitions. The theorem `Good_held_for in` uses the recursive definition of the Held_For operator while the last theorem involve the "set" definition.

```
simple  : THEORY
BEGIN
K: posreal = 100
IMPORTING Held_For[K]
t, t1, t2: VAR clock
param_trip: timed_cond = (LAMBDA (t):
     IF (t<1000) THEN FALSE ELSE TRUE ENDIF)

duration:posreal = 295
clock_lt_le: lemma t1 < t2 => t1 <= t2 - K

Good_held_for: THEOREM (t>=1000+duration) IMPLIES
        Held_For(param_trip,duration)(t)

Good_held_for_set: THEOREM (t>=1000+duration) IMPLIES
        Held_For_set(param_trip,duration)(t)
 END simple
```

In order to see the difference in the run time resulted from using the two definitions to finish the similar proofs, we provide 4 pairs of *loop time K* and *duration* to compare the proofs of the theorems `Good_Held_for` and `Good_Held_for_set`.

The theorem proofs were run under SUN 450 Sever (Quad Processor) with four 300 MHz CPU and 1 GB RAM. The PVS version is 2.2. The run time is recorded in Figure A.1.

			Run Time (seconds)	
K	duration	duration/K	Held_For	Held_For_set
100	100	1	0.96	0.67
50	295	5.9	1.62	0.60
20	4000	200	19.93	0.61
10	5000	500	62.73	0.75

Figure A.1: Comparison results of 2 Held_For definitions

As we see, the larger the ratio of duration/K is, the more run time the recursive definition needs to accomplish the proof. For the "set" version, there is no significant increase in the run time consumed. The reason is simple. For the recursive definition of the Held_For operator, a large amount of time is spent to "unspool" the definition. In this case, the PVS proof script is huge, consisting mainly of definition rewriting.

The "set" version, however, involves almost the same workload although *duration/K* is different. Table A.1 shows the proof commands using "set" version of Held_For for two different parameter settings. The left side shows the proof com-

Table A.1: Proof command comparison for Held_For "set" definition

`(SKOSIMP)`	`(SKOSIMP)`
`(EXPAND "Held_For_set")`	`(EXPAND "Held_For_set")`
`(INST 1 1000)`	`(INST 1 1000)`
`(("1" (GRIND))`	`(("1" (GRIND))`
`("2" (INST 1 20) (GRIND)))`	`("2" (INST 1 100) (GRIND)))`

mands when *K* and *duration* are 100 and 100 respectively; whereas the right side shows the commands when K and *duration* are 10 and 5000 respectively. The proof commands are almost the same except the instantiated values in the last row.

Considering the case when *duration/K = 500*, the recursive definition requires more than 1 minute run time while the "set" version uses less than 1 second. The case will get worse when *duration/K* is even bigger. We can conclude that the "set" definition can handle the real-time properties spanning "large" durations more easier than the recursive one.

C The PVS Theory `timing_lemma`

This theory states and proves some PVS lemmas regarding the timing properties. These lemmas are "pulled out" from the actual theorem proofs and can be reused to reduce the proving effort in the future.

C.1 PVS specification for the `timing_lemma` theory

In this subsection we present 11 lemmas regarding the intersample time values, the $Held\_For$ operator and the relationship between the loop time and the duration.

```
Timing_lemmas [ K: posreal ]: THEORY
 BEGIN
 IMPORTING Clocks[K]
 t, t1, t2: VAR clock
 duration: VAR time
 m, n:VAR nat

 le_inner_clockvar: LEMMA
      FORALL (k_i:{k: posreal|(t1<k) & (k<t1+K)}): (t2<=k_i) => (t2<=t1)
 ge_inner_clockvar: LEMMA
      FORALL (k_i:{k: posreal|(t1<k) & (k<t1+K)}): (t2>=k_i) => (t2>=t1+K)
 lt_le_clockvar: LEMMA t1 < t2 => t1  <= pre(t2)
 gt_ge_clockvar: LEMMA t1 > t2 => t1 >= next(t2)
 eq_inner_clockvar: LEMMA  FORALL (t1, t2: clock):
           (t1-K  <= t2 and t2 <=t1) IMPLIES
                t2=t1- K or t2 = t1
 eq_inner3_clockvar: LEMMA FORALL (t1, t2):
```

```
           (t1-2*K  <= t2 and t2 <=t1) IMPLIES
             t2=t1-2*K or  t2=t1-K or t2 = t1
 zero_inner_clockvar: LEMMA
           FORALL (t: clock): t< K => t=0
 loop_duration1: LEMMA FORALL (n: posnat):
        K >= duration IMPLIES  n*K >= duration
 loop_duration_n: LEMMA EXISTS n: n*K >= duration
 Held_For_next: lemma forall (t:noninit_elem):
        tcond(t) AND Held_For(tcond, duration)(pre(t))
      IMPLIES Held_For(tcond, duration)(t)
 Held_For_set_next: lemma forall (t:noninit_elem):
        tcond(t) AND Held_For_set(tcond, duration)(pre(t))
      IMPLIES Held_For_set(tcond, duration)(t)
 END Timing_lemmas
```

C.2 The PVS proofs of the timing_lemma theory

Below are the proof scripts for the above timing lemmas:

le_inner_clockvar:

```
(""
 (SKOSIMP)
 (TYPEPRED "t2!1" "t1!1" "k_i!1")
 (GRIND)
 (CASE "n!2=n!1")
 (("1" (GRIND))
  ("2"
   (BOTH-SIDES "/" K -11)
   (BOTH-SIDES "/" K -10)
   (BOTH-SIDES "/" "K" 2)
   (GROUND))))
```

ge_inner_clockvar:

```
(""
 (SKOSIMP)
 (TYPEPRED "t2!1" "t1!1" "k_i!1")
 (GRIND)
 (CASE "n!2=n!1")
 (("1" (GRIND))
  ("2"
   (LEMMA "div_cancel2")
   (BOTH-SIDES "/" "K" -11)
   (BOTH-SIDES "/" "K" -12)
   (BOTH-SIDES "/" "K" -10)
   (BOTH-SIDES "/" "K" 2)
   (INST -1 "K" "n!1")
   (GROUND))))
```

lt_le_clockvar:

```
(""
 (SKOLEM-TYPEPRED)
 (GRIND)
 (BOTH-SIDES / K -3)
 (BOTH-SIDES / K -5)
 (BOTH-SIDES / K -7)
 (BOTH-SIDES / K 1)
 (GRIND))
```

gt_ge_clockvar:

```
(""
 (SKOLEM-TYPEPRED)
 (GRIND)
 (BOTH-SIDES / K -3)
 (BOTH-SIDES / K -5)
 (BOTH-SIDES / K -7)
 (BOTH-SIDES / K 1)
 (GRIND))
```

eq_inner_clockvar:

```
(""
 (GRIND)
 (BOTH-SIDES "/" "K" -7)
 (BOTH-SIDES "/" "K" -8)
 (BOTH-SIDES "/" "K" -1)
 (BOTH-SIDES "/" "K" 1)
 (("1" (BOTH-SIDES "/" "K" 2) (GRIND)) ("2" (GRIND))))
```

eq_inner3_clockvar:

```
(""
 (GRIND)
 (BOTH-SIDES "/" "K" -7)
 (BOTH-SIDES "/" "K" -8)
 (BOTH-SIDES "/" "K" 1)
 (("1"
   (BOTH-SIDES "/" "K" 2)
   (("1" (BOTH-SIDES "/" "K" 3) (GRIND)) ("2" (GRIND))))
  ("2" (GRIND))))
```

zero_inner_clockvar:

```
(""
 (SKOLEM-TYPEPRED)
 (GROUND)
 (GRIND)
 (BOTH-SIDES / K -4)
 (("1" (BOTH-SIDES / K 1) (BOTH-SIDES / K -2) (GRIND))
  ("2"
   (GRIND)
   (LEMMA "both_sides_times_pos_lt1")
   (INST -1 K "n!1" 1)
   (GRIND))))
```

loop_duration1:

```
(""
 (GRIND)
 (TYPEPRED "K")
 (LEMMA "ge_times_ge_pos")
 (INST -1 "duration!1" 1 "n!1" "K")
 (GRIND))
```

loop_duration_n:

```
(""
 (SKOLEM-TYPEPRED)
 (TYPEPRED "K")
 (INST 1 "floor(duration!1/K)+1")
 (CASE "floor(duration!1 / K)+1 >=duration!1 / K")
 (("1" (GRIND) (BOTH-SIDES "*" "K" -1) (("1" (GRIND)) ("2" (GRIND))))
  ("2" (GRIND))))
```

Held_For_set_next:

```
(""
 (SKOLEM-TYPEPRED)
 (GROUND)
 (EXPAND "Held_For\_set")
 (SKOSIMP*)
 (INST 2 "t!2")
 (GROUND)
 (("1" (GRIND))
  ("2"
   (SKOLEM-TYPEPRED)
   (INST -10 "t_i!1")
   (CASE "t_i!1=t!1")
   (("1" (GRIND))
    ("2"
     (CASE "t_i!1=pre(t!1)")
     (("1" (GRIND))
      ("2"
       (GRIND)
       (BOTH-SIDES "/" "K")
       (("1"
         (HIDE -10 4)
         (BOTH-SIDES "/" "K" -4)
         (BOTH-SIDES "/" "K" -5)
         (BOTH-SIDES "/" "K" 3)
         (GRIND))
        ("2"
         (HIDE 5)
         (TYPEPRED "t!2")
(SKOSIMP)
(SIMPLIFY 1)
(REPLACE -2 *)
(HIDE -2 -6 -12 -3)
(BOTH-SIDES "/" "K" 2)
(BOTH-SIDES "/" "K" 3)
(BOTH-SIDES "/" "K" 1)
(("1"
  (BOTH-SIDES "/" "K" -9)
  (("1"
    (BOTH-SIDES "/" "K" -4)
    (BOTH-SIDES "/" "K" -5)
    (GRIND))
   ("2" (GRIND))))
 ("2"
  (REVEAL -2)
  (GRIND)
  (TYPEPRED "K")
  (BOTH-SIDES "*" "K" 3)
  (GRIND))))))))))))
```

D The PVS Theory simple

This section presents the PVS theory simple. The major purpose of this theory is to illustrate the use of the PVS Held_For theory and to compare the two definitions of the *Held_For* operator.

D.1 The PVS specification for the simple theory

```
simple  : THEORY
BEGIN
K: posreal = 100
IMPORTING Held_For[K]
t, t1, t2: VAR clock
param_trip: timed_cond = (LAMBDA (t):
     IF (t<1000) THEN FALSE ELSE TRUE ENDIF)
duration:posreal = 295
clock_lt_le: lemma t1 < t2 => t1 <= t2 - K
```

```
% The following theorems involved the recursive definition  of Held_For Operator
  Good_Held_for: THEOREM (t>=1000+duration) IMPLIES
        Held_For(param_trip,duration)(t)
  Bad_held_for: THEOREM (t>=1000+duration-K) IMPLIES
        Held_For(param_trip,duration)(t)
  Not_bad_held_for: THEOREM  EXISTS (t: clock[K]):
       not  ((t>=1000+duration-K) IMPLIES
             Held_For(param_trip,duration)(t))

% The following theorems involved "set" definition  of Held_For Operator
  Good_Held_for_set: THEOREM (t>=1000+duration) IMPLIES
       Held_For_set(param_trip,duration)(t)
  Bad_held_for_set: THEOREM (t>=1000+duration-K) IMPLIES
       Held_For_set(param_trip,duration)(t)
  Not_bad_held_for_set: THEOREM  EXISTS (t: clock[K]):
       not  ((t>=1000+duration-K) IMPLIES
            Held_For_set(param_trip,duration)(t))
 END simple
```

D.2 The PVS proofs of the `simple` Theory

In this subsection we present proof scripts for the 6 theorems in the `simple` theory.

The theorem `Good_Held_for`:

```
(""
 (SKOLEM!)
 (GRIND)
 (EXPAND "heldfor")
 (BDDSIMP)
 (TYPEPRED "t!1")
 (EXPAND "param_trip")
 (BDDSIMP 1)
 (GRIND))
```

The theorem `Bad_held_for`:

```
(""
 (GRIND)
 (EXPAND "heldfor")
 (GRIND)
 (("1" (EXPAND "heldfor") (BDDSIMP) (POSTPONE))
  ("2" (POSTPONE))))
```

The theorem `Not_bad_held_for`**:**

```
("" (INST 1 1200)
    (("1" (GRIND)) ("2" (INST 1 12) (GRIND))))
```

The theorem `Good_Held_for_set`**:**

```
(""
 (SKOSIMP)
 (EXPAND "Held_For_set")
 (INST 1 1000)
 (("1" (GRIND)) ("2" (INST 1 10) (GRIND))))
```

The theorem `Bad_held_for_set`**:**

```
(""
 (SKOSIMP)
 (EXPAND "Held_For_set")
 (INST 1 1000)
 (("1" (GRIND) (POSTPONE)) ("2" (INST 1 10) (GRIND))))
```

The theorem `Not_bad_held_for_set`**:**

```
(""
 (INST 1 1200)
 (("1"
   (GROUND)
   (("1"
     (EXPAND "Held_For_set")
     (SKOSIMP)
     (INST -2 900)
     (("1" (GRIND))
      ("2"
       (GROUND)
      (("1" (INST 1 9) (GRIND))
       ("2"
        (EXPAND "duration")
        (BOTH-SIDES + "t!1-295" -1)
        (ASSERT)
        (TYPEPRED "t!1")
        (LEMMA "le_inner_clockvar")
        (INST -1 900 "t!1" 905)
        (("1" (GRIND)) ("2" (GRIND))
         ("3" (INST 1 9) (GRIND))))))))
   ("2" (GRIND))))
 ("2" (INST 1 12) (GRIND))))
```

E The PVS Theory `sealed_in`

In this section we present the PVS specification and proofs for the channel trip sealed-in timing block.

E.1 The PVS specification for the `sealed_in` theory

This subsection provides 4 PVS theories with respect to the specification of the channel trip sealed-in block. Among them, the theory `sealed_in` states the original definitions of the SRS and SDD. As discussed in Chapter 4, lots of unprovable sequents are encountered when using these definitions to prove the main block comparison theorem. The theory `sealed_in2` fixed the errors mentioned in Section 4.3.1-3 of the thesis but it still has some problems in the SDD function. The theory `sealed_in3` fixed all of the errors and the main block comparison theorem in `sealed_in3` has been proved successfully. The theory `sealed_in_safe` present the “safer” version of the SRS and SDD where the manual reset is not allowed when there is a parameter trip.

```
sealed_in : THEORY
BEGIN

loop_time: nat =100
IMPORTING Held_For[loop_time]
```

```
  t: VAR clock
  timed_cond: TYPE = [clock -> bool]
  param_trip: VAR timed_cond
  manual_reset: VAR timed_cond
  seal_in_delay:nat = 150

 % The original SRS specification of channel trip sealed-in function
  c_sealed_in(param_trip,manual_reset)(t:clock): RECURSIVE bool = COND
    init(t)  ->    TRUE,
    NOT init(t) AND  Held_For(param_trip,seal_in_delay)(t) -> TRUE,
    NOT init(t) AND  NOT Held_For(param_trip,seal_in_delay)(t)
         AND manual_reset(t) -> FALSE,
    NOT init(t) AND  NOT Held_For(param_trip,seal_in_delay)(t) AND
         NOT manual_reset(t) -> c_sealed_in(param_trip,manual_reset)(pre(t))
      ENDCOND
    MEASURE rank(t)

  true(b):bool = b
  false(b):bool = not b
  DC(b):bool = true  % DC : Don't Care

% The original SRS expressed in TABLE and X operator
c_sealed_in_table(t): RECURSIVE bool =

  LET X = (LAMBDA (a: pred[bool]),
                  (b: pred[bool]),
                  (c: pred[bool]):
                   a(init(t)) &
                   b(Held_For(param_trip,seal_in_delay)(t)) &
                   c(manual_reset(t)))

  IN TABLE

%    init(t)
%       |    Held_For(param_trip,seal_in_delay)(t)
%       |        |   manual_reset(t)
%       |        |      |
%       v        v      v      Channel trip sealed_in
  %----------|-------|--------|--------------------%
  | X(true ,     DC   ,  DC)   |     true           ||
  %----------|--------|-------|---------------------%
  | X(false ,  true   ,  DC)   |     true           ||
  %----------|--------|-------|----------------------%
  | X(false ,  false,   true) |     false          ||
  %----------|--------|-------|----------------------%
  | X(false,   false,   false)|  c_sealed_in(pre(t))||
  %----------|--------|-------|----------------------%

 ENDTABLE

 MEASURE rank(t)

  % The original SDD implementation of channel trip sealed-in function
  KSEAL: nat = 150
  Channel_state: TYPE = {NCHNT, CHNT, SEAL}
  SDD_State: TYPE = [# CHNTS1: Channel_state, l_SealDly: clock #]

  ECHNT(PT:bool,MR:bool,l_SealDly_s1:clock,CHNTS1_s1:Channel_state): SDD_State = COND
    NOT PT AND CHNTS1_s1=SEAL AND MR ->
           (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
    NOT PT AND CHNTS1_s1=SEAL AND NOT MR ->
           (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    NOT PT AND NOT CHNTS1_s1=SEAL ->
           (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
    PT AND l_SealDly_s1=0   ->
           (# CHNTS1:= CHNT, l_SealDly:= next(0) #),
    PT AND 0<l_SealDly_s1 AND l_SealDly_s1<KSEAL ->
           (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
    PT AND l_SealDly_s1>=KSEAL ->
            (# CHNTS1:= SEAL, l_SealDly:= 0 #)
    ENDCOND

   ICHNT: SDD_State = (# CHNTS1:= CHNT, l_SealDly:= 0 #)
   sdd(param_trip,manual_reset)(t): RECURSIVE SDD_State = COND
      init(t) -> ICHNT,
      ELSE   -> ECHNT(param_trip(t), manual_reset(t),
   l_SealDly(sdd(param_trip,manual_reset)(pre(t))),
   CHNTS1(sdd(param_trip,manual_reset)(pre(t))))
```

```
    ENDCOND
    MEASURE rank(t)

  % The SRS in COND corresponds to that in  TABLE + X operator
   Seal_cond_eq_table: THEOREM c_sealed_in = c_sealed_in_table

  % Specify main block comparison theorem
  Seal: THEOREM c_sealed_in(param_trip,manual_reset)(t) =
                SEAL?(CHNTS1(sdd(param_trip,manual_reset)(t))
  END sealed_in

sealed_in2  : THEORY
  BEGIN
  loop_time: nat =100
  IMPORTING  sealed_in_text
  t: VAR clock
  timed_cond: TYPE = [clock -> bool]
  param_trip: VAR timed_cond
  manual_reset: VAR timed_cond
  seal_in_delay:nat = 150
  KSEAL: nat = 150

  SDD_State: TYPE = [#CHNTS1: Channel_state, l_SealDly: clock #]

  ECHNT2(PT:bool,MR:bool,l_SealDly_s1:clock,CHNTS1_s1:Channel_state): SDD_State = COND
    NOT PT AND CHNTS1_s1=SEAL AND MR ->
        (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
    NOT PT AND CHNTS1_s1=SEAL AND NOT MR ->
        (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    NOT PT AND NOT CHNTS1_s1=SEAL ->
        (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
    PT AND l_SealDly_s1=0  ->
        (# CHNTS1:= CHNT, l_SealDly:= next(0) #),
    PT AND 0<l_SealDly_s1 AND  l_SealDly_s1<KSEAL ->
        (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
    PT AND l_SealDly_s1>=KSEAL ->
        (# CHNTS1:= SEAL, l_SealDly:= l_SealDly_s1#)
    ENDCOND
```

```
sdd2(param_trip,manual_reset)(t): RECURSIVE SDD_State = COND
   init(t)  AND NOT param_trip (t) ->
         (# CHNTS1:= SEAL, l_SealDly:= 0 #),
   init(t)   AND     param_trip (t) ->
         (# CHNTS1:= SEAL, l_SealDly:= next(0) #),

   NOT init(t)  -> ECHNT2(param_trip(t), manual_reset(t),
         l_SealDly(sdd2(param_trip,manual_reset)(pre(t))),
         CHNTS1(sdd2(param_trip,manual_reset)(pre(t))))
   ENDCOND
   MEASURE rank(t)

 Seal2: THEOREM
       c_sealed_in(param_trip,manual_reset)(t) =
           SEAL?(CHNTS1(sdd2(param_trip,manual_reset)(t)))
 END sealed_in2

sealed_in3  : THEORY
  BEGIN
   loop_time: nat =100
   IMPORTING  sealed_in

   t: VAR clock
   timed_cond: TYPE = [clock -> bool]
   param_trip: VAR timed_cond
   manual_reset: VAR timed_cond
   seal_in_delay:nat = 150
   KSEAL: nat = 150
   SDD_State: TYPE = [#CHNTS1: Channel_state, l_SealDly: clock #]

ECHNT3(PT:bool,MR:bool,l_SealDly_s1:clock,CHNTS1_s1:Channel_state): SDD_State = COND
   NOT PT AND CHNTS1_s1=SEAL AND MR ->
       (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
   NOT PT AND CHNTS1_s1=SEAL AND NOT MR ->
       (# CHNTS1:= SEAL, l_SealDly:= 0 #),
   NOT PT AND NOT CHNTS1_s1=SEAL ->
       (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
   PT AND l_SealDly_s1 < KSEAL AND MR  ->
       (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
    PT AND l_SealDly_s1 < KSEAL AND NOT MR AND  CHNTS1_s1=SEAL ->
        (# CHNTS1:= SEAL, l_SealDly:= next(l_SealDly_s1) #),
    PT AND l_SealDly_s1<KSEAL AND NOT MR AND NOT CHNTS1_s1 = SEAL ->
        (# CHNTS1:= CHNT, l_SealDly:= next(l_SealDly_s1) #),
    PT AND l_SealDly_s1>=KSEAL ->
        (# CHNTS1:= SEAL, l_SealDly:= l_SealDly_s1#)
    ENDCOND

 sdd3(param_trip,manual_reset)(t): RECURSIVE SDD_State = COND
    init(t)  AND NOT param_trip (t) ->
        (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    init(t)  AND param_trip (t) ->
        (# CHNTS1:= SEAL, l_SealDly:= next(0) #),
    NOT init(t)  ->
        ECHNT3(param_trip(t), manual_reset(t),
            l_SealDly(sdd3(param_trip,manual_reset)(pre(t))),
            CHNTS1(sdd3(param_trip,manual_reset)(pre(t))))
    ENDCOND
    MEASURE rank(t)

  Seal3: THEOREM c_sealed_in(param_trip,manual_reset)(t) =
                 SEAL?(CHNTS1(sdd3(param_trip,manual_reset)(t)))
  END sealed_in3

sealed_in_safe : THEORY

  BEGIN
  loop_time: nat = 100
  IMPORTING Held_For[loop_time]

  t: VAR clock
  timed_cond: TYPE = [clock -> bool]
  param_trip: timed_cond
  manual_reset: timed_cond
  seal_in_delay:nat = 150

 % SRS - ``safe''  version of  Param_trip lock  function
 c_sealed_in2(t): RECURSIVE bool = COND
    init(t) -> true,
    NOT init(t) AND Held_For( param_trip,seal_in_delay)(t) -> true,
    NOT init(t) AND NOT Held_For( param_trip,seal_in_delay)(t)
      AND manual_reset(t) AND  param_trip(t)->c_sealed_in2(pre(t)),
    NOT init(t) AND NOT Held_For( param_trip,seal_in_delay)(t)
      AND manual_reset(t) AND NOT  param_trip(t)-> false,
    NOT init(t) AND NOT Held_For( param_trip,seal_in_delay)(t) AND
      NOT manual_reset(t) ->c_sealed_in2(pre(t))
    ENDCOND
    MEASURE rank(t)

% SDD - ``Safer'' version of the SDD Function
  Channel_State: TYPE = {CHNT, NCHNT, SEAL}

  SDD_State: TYPE = [#
CHNTS1: Channel_State,
        l_SealDly: clock #]

  sdd4(t): RECURSIVE SDD_State = COND
    init(t) AND NOT  param_trip(t) ->
          (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    init(t) AND  param_trip(t) ->
          (# CHNTS1:= SEAL, l_SealDly:= next(0) #),
    NOT init(t) AND NOT  param_trip(t) AND SEAL?(CHNTS1 (sdd4(pre(t))))
AND manual_reset(t) ->
  (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
    NOT init(t) AND NOT  param_trip(t) AND SEAL?(CHNTS1(sdd4(pre(t))))
AND NOT manual_reset(t) ->
  (# CHNTS1:= SEAL, l_SealDly:= 0 #),
    NOT init(t) AND NOT  param_trip(t) AND NOT SEAL?(CHNTS1(sdd4(pre(t)))) ->
  (# CHNTS1:= NCHNT, l_SealDly:= 0 #),
    NOT init(t) AND  param_trip(t) AND l_SealDly(sdd4(pre(t)))<seal_in_delay
        AND NOT SEAL?(CHNTS1(sdd4(pre(t))))  ->
  (#CHNTS1:=CHNT, l_SealDly:=next(l_SealDly(sdd4(pre(t))))#),
    NOT init(t) AND  param_trip(t) AND l_SealDly(sdd4(pre(t)))< seal_in_delay
        AND SEAL?(CHNTS1(sdd4(pre(t))))  ->
  (#CHNTS1:=SEAL, l_SealDly:=next(l_SealDly(sdd4(pre(t))))#),
    NOT init(t) AND  param_trip(t) AND l_SealDly(sdd4(pre(t)))>=seal_in_delay ->
  (#CHNTS1:=SEAL,l_SealDly:= l_SealDly(sdd4(pre(t)))#)
    ENDCOND
    MEASURE rank(t)

   % Specify  main theorem and prove using the general clock induction
  Seal_eq_sdd4: THEOREM
     c_sealed_in2(t) = SEAL?(CHNTS1(sdd4(t)))
  % Specify  main theorem and prove using strong induction
  Seal_eq_sdd_strong_induct: THEOREM
     c_sealed_in2(t) = SEAL?(CHNTS1(sdd4(t)))
  END sealed_in_safe
```

E.2 The PVS proofs for the channel trip sealed-in block

In this subsection, we provide the proof scripts for the main block comparison theorems `Seal3`, `Seal_eq_sdd4` and `Seal_eq_sdd_strong_induct` in the PVS theories `sealed_in3` and `sealed_in_safe`. In the theory `sealed_in` the SRS function is presented in COND construct as well as in TABLE construct with *X operator*. We show these two expressions are equivalent by proving the theorem `Seal_cond_eq_table` at the end of this subsection.

The theorem `Seal3`:

```
("" 
(INDUCT "t" 1 "clock_induction")
(("1" (SKOSIMP*) (GRIND) (HIDE -2) (EXPAND "sdd3") (GRIND))
 ("2"
  (SKOSIMP)
  (SKOSIMP)
  (INST -1 "manual_reset!1" "param_trip!1")
  (BDDSIMP)
  (("1"
    (EXPAND "c_sealed_in" -3)
    (ASSERT)
    (BDDSIMP)
    (("1"
      (GRIND)
      (EXPAND "heldfor")
      (GRIND)
      (EXPAND "heldfor")
```

```
     (GRIND)
     (("1" (EXPAND "heldfor") (GRIND)) ("2" (EXPAND "heldfor") (GRIND))))
    ("2"
     (EXPAND "sdd3" -2)
     (LIFT-IF)
     (ASSERT)
     (BDDSIMP)
     (("1"
       (GRIND)
       (("1" (EXPAND "sdd3") (GRIND))
        ("2" (HIDE -1 -2 -3 -4 1) (EXPAND "sdd3") (ASSERT))))
      ("2"
       (EXPAND "ECHNT3")
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
       (("1" (ASSERT) (GRIND))
        ("2" (HIDE -1) (EXPAND "sdd3" 3) (EXPAND "ECHNT3") (GRIND))))))))
  ("2"
   (EXPAND "sdd3" -3)
   (EXPAND "ECHNT3")
   (GRIND)
   (EXPAND "heldfor")
   (BDDSIMP)
   (("1"
     (GRIND)
     (BOTH-SIDES + 100 -3)
     (("1"
       (ASSERT)
       (REPLACE -3 *)
       (EXPAND "c_sealed_in")
       (ASSERT)
       (EXPAND "sdd3" -1)
       (ASSERT)
       (GRIND)
       (GRIND)
       (EXPAND "sdd3")
       (GRIND))
      ("2" (ASSERT))))
      ("2"
       (EXPAND "heldfor")
       (GRIND)
       (HIDE -2)
       (EXPAND "c_sealed_in")
       (GRIND)
       (("1" (HIDE -1 -2 1) (GRIND) (EXPAND "sdd3") (GRIND))
        ("2" (EXPAND "heldfor") (GRIND))
        ("3" (HIDE -1) (EXPAND "sdd3") (GRIND) (EXPAND "sdd3") (GRIND))))
      ("3" (EXPAND "sdd3") (GRIND))))
    ("3"
     (EXPAND "sdd3" 3)
     (EXPAND "ECHNT3")
     (LIFT-IF)
     (ASSERT)
     (BDDSIMP)
     (("1"
       (GRIND)
       (EXPAND "heldfor")
       (GRIND)
       (EXPAND "heldfor")
       (GRIND)
       (("1" (EXPAND "heldfor") (GRIND)) ("2" (EXPAND "heldfor") (GRIND))
        ("3" (EXPAND "heldfor") (GRIND))))
      ("2" (GRIND))))
    ("4"
     (EXPAND "sdd3" -1)
     (EXPAND "ECHNT3")
     (GRIND)
     (EXPAND "heldfor")
     (GRIND)
     (("1"
       (EXPAND "heldfor")
       (GRIND)
       (EXPAND "sdd3" 2)
       (GRIND)
       (("1"
         (EXPAND "heldfor")
         (GRIND)
         (EXPAND "heldfor")
         (GRIND)
         (EXPAND "sdd3")
         (GRIND))
        ("2" (EXPAND "sdd3") (GRIND)) ("3" (EXPAND "sdd3") (GRIND))))
      ("2" (EXPAND "sdd3") (GRIND))))))
  ("3" (GRIND)) ("4" (GRIND))))
```

The theorem: `Seal_eq_sdd4`:

```
(""
 (INDUCT "t" 1 "clock_induction")
 (("1" (GRIND) (EXPAND* "sdd4" "init"))
  ("2"
   (SKOSIMP)
   (BDDSIMP)
   (("1" (EXPAND "sdd4" 1) (LIFT-IF) (ASSERT) (BDDSIMP) (GRIND))
    ("2" (GRIND))
    ("3"
     (EXPAND " c_sealed_in2" -1)
     (ASSERT)
     (BDDSIMP)
     (EXPAND "sdd4" 3)
     (LIFT-IF)
     (GRIND)
     (EXPAND "heldfor")
     (GRIND)
     (EXPAND "sdd4" 3)
     (GRIND))
    ("4"
     (EXPAND "sdd4" -1)
     (LIFT-IF)
     (ASSERT)
     (BDDSIMP)
     (EXPAND " c_sealed_in2" 3)
     (BDDSIMP)
     (EXPAND "sdd4" 4)
     (LIFT-IF)
     (ASSERT)
     (GRIND)
     (EXPAND "sdd4" 6)
     (GRIND))))
  ("3" (GRIND))))
```

The theorem: `Seal_eq_sdd_strong_induct:`

```
(""
 (INDUCT "t" 1 "clock_strong_induction2")
 (("1"
   (SKOSIMP*)
   (INST -1 "pre(t!1)")
   (("1"
     (BDDSIMP)
     (("1"
       (EXPAND " c_sealed_in2" -3)
       (ASSERT)
       (BDDSIMP)
       (("1" (EXPAND "sdd4") (ASSERT))
        ("2" (GRIND) (EXPAND "heldfor") (GRIND)) ("3" (GRIND))
        ("4" (EXPAND "sdd4" 2) (LIFT-IF) (ASSERT))))
      ("2" (EXPAND " c_sealed_in2" 1) (BDDSIMP) (GRIND))
      ("3"
       (EXPAND " c_sealed_in2" -1)
       (ASSERT)
       (BDDSIMP)
       (("1" (HIDE 1 2) (EXPAND "sdd4") (GRIND))
        ("2"
         (EXPAND "sdd4" 3)
         (LIFT-IF)
         (GRIND)
         (EXPAND "heldfor")
         (GRIND)
         (EXPAND "sdd4" -5)
         (GRIND))))
      ("4"
       (EXPAND "sdd4" -1)
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
       (("1" (EXPAND " c_sealed_in2" 3) (BDDSIMP))
        ("2"
         (EXPAND " c_sealed_in2" 3)
         (BDDSIMP)
         (EXPAND "sdd4" 5)
         (GRIND)
         (EXPAND "sdd4" 7)
         (GRIND))))
      ("5" (GRIND)) ("6" (GRIND))))
    ("2" (GRIND)) ("3" (GRIND) (EXPAND "sdd4") (GRIND))))
  ("2" (GRIND))))
```

The theorem `Seal_cond_eq_table`**:**

```
(""
 (APPLY-EXTENSIONALITY)
 (("1"
   (GRIND)
   (("1" (EXPAND "heldfor") (GRIND))
    ("2" (EXPAND* "c_sealed_in" "c_sealed_in_table") (GRIND))))
  ("2" (EXPAND "loop_time") (PROPAX))))
```

F The PVS Theory `ChanReset`

In this section we present the PVS specification and proofs for the channel trip reset timing block.

F.1 The PVS specification for the `ChanReset` theory

This subsection states the "fixed" PVS specification of the channel trip reset block. The SRS functions in this theory are the second version labeled *f_ChanReset2* and *srs2* whereas the SDD functions below are the third version labelled *crk_chan_reset_s3* and *sdd3*.

```
ChanReset: THEORY
```

```
BEGIN
IMPORTING Held_For[50]
t: VAR clock
% Constants
tim_stuck_c: int = 17 %ms
k_Debounce: int = 50 %ms
k_Stuck: int = 150 %ms
gbl_word_t: TYPE = subrange(0, 65535)
% SRS Types
t_Reset: TYPE = {e_ClearTrip, e_WaitForOff, e_WaitForOn, e_ResetStuck}
t_Key: TYPE = {e_KeyOn, e_NotKeyOn}
t_SrsInput: TYPE = [clock -> t_Key]

%  SRS Function: f_ChanReset2
m_ChanReset: VAR {m: t_SrsInput| m(0) = e_NotKeyOn}
f_ChanReset2(m_ChanReset, f_ChanReset_si:t_Reset, t1:clock): t_Reset =
    IF  Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Stuck)(t1)
    THEN  e_ResetStuck
    ELSE
       COND
        Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
        NOT Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
         COND
           e_WaitForOn?(f_ChanReset_si)  ->  e_WaitForOff,
           e_WaitForOff?(f_ChanReset_si) ->  e_WaitForOff,
           e_ResetStuck?(f_ChanReset_si) ->  e_ResetStuck,
           e_ClearTrip? (f_ChanReset_si) ->  e_WaitForOff
         ENDCOND,

        NOT Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
          Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
         COND
           e_WaitForOn? (f_ChanReset_si) ->  e_WaitForOn,
           e_WaitForOff?(f_ChanReset_si) ->  e_ClearTrip,
           e_ResetStuck?(f_ChanReset_si) ->  e_WaitForOn,
           e_ClearTrip? (f_ChanReset_si) ->  e_WaitForOn
         ENDCOND,

        NOT Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Debounce)(t1) &
```

```
           NOT Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t1) ->
         COND
           e_WaitForOn? (f_ChanReset_si) ->  e_WaitForOn,
           e_WaitForOff?(f_ChanReset_si) ->  e_WaitForOff,
           e_ResetStuck?(f_ChanReset_si) ->  e_ResetStuck,
           e_ClearTrip? (f_ChanReset_si) ->  e_WaitForOn
         ENDCOND
       ENDCOND
    ENDIF

srs2(m_ChanReset)(t): RECURSIVE t_Reset =
  IF init(t) THEN e_WaitForOn
  ELSE f_ChanReset2(m_ChanReset,srs2(m_ChanReset)(pre(t)),t)
  ENDIF
  MEASURE rank(t)

%  The SDD Types
crk_reset_t2: TYPE = {
   wait_for_on_e,
   wait_for_off_e,
   reset_stuck_e,
   clear_trip_e }

dip_key_t: TYPE = {keyon_e, keyoff_e}
sdd_state_t2: TYPE = [#tim_c_key_s:gbl_word_t,  crk_chan_reset_s:crk_reset_t#]

%  The SDD function
crk_chan_reset_s3(
   tim:   gbl_word_t,
   mcr:   dip_key_t,
   crk_v: dip_key_t,
   crk2:  crk_reset_t): crk_reset_t = COND
      (tim <=1) -> reset_stuck_e,
      NOT(tim <=1) & (mcr = keyon_e) & (crk_v = keyon_e) ->
            COND
                (crk2 = wait_for_on_e)  -> wait_for_off_e,
                (crk2 = wait_for_off_e) -> wait_for_off_e,
                (crk2 = reset_stuck_e)  -> reset_stuck_e,
                (crk2 = clear_trip_e)  ->  wait_for_off_e
            ENDCOND,
      NOT(tim <=1) & (mcr = keyon_e) & NOT (crk_v = keyon_e) ->
            COND
                (crk2 = wait_for_on_e)  -> wait_for_on_e,
                (crk2 = wait_for_off_e) -> wait_for_off_e,
                (crk2 = reset_stuck_e)  -> reset_stuck_e,
                (crk2 = clear_trip_e)   -> wait_for_on_e
            ENDCOND,

      NOT (tim <=1) & NOT (mcr = keyon_e) & (crk_v = keyon_e) ->
            COND
                (crk2 = wait_for_on_e)  -> wait_for_on_e,
                (crk2 = wait_for_off_e) -> wait_for_off_e,
                (crk2 = reset_stuck_e)  -> reset_stuck_e,
            (crk2 = clear_trip_e)  ->  wait_for_on_e
            ENDCOND,

      NOT(tim <=1) & NOT (mcr = keyon_e) & NOT (crk_v = keyon_e) ->
            COND
                (crk2 = wait_for_on_e)  -> wait_for_on_e,
                (crk2 = wait_for_off_e) -> clear_trip_e,
                (crk2 = reset_stuck_e)  -> wait_for_on_e,
                (crk2 = clear_trip_e)   -> wait_for_on_e
            ENDCOND
      ENDCOND

%   DEFINITION OF SDD STATE MACHINE

% Abstraction Function: Abst_Key
Abst_Key(x: t_Key): dip_key_t =
   COND
       (x = e_KeyOn)    -> keyon_e,
       (x = e_NotKeyOn) -> keyoff_e
   ENDCOND

% Abst_pre_key, added initial value
Abst_pre_key(m_ChanReset)(t): dip_key_t =
   COND
       init(t) -> keyoff_e,
       else    -> Abst_Key(m_ChanReset(t))
   ENDCOND

% countdown timer: tim_c_key_s2
tim_c_key_s2(x: dip_key_t, tim_c_key_s_si: gbl_word_t): gbl_word_t =
    COND
      (x = keyon_e) -> max(0, tim_c_key_s_si-4),
      ELSE          -> tim_stuck_c
   ENDCOND

% sdd3() function
sdd3(m_ChanReset)(t): RECURSIVE sdd_state_t2 =
  COND
   init(t) -> (#tim_c_key_s:=tim_stuck_c,
               crk_chan_reset_s:=wait_for_on_e #),
   ELSE    -> (#tim_c_key_s:=
                 tim_c_key_s2(Abst_Key(m_ChanReset(t)),
                     tim_c_key_s(sdd3(m_ChanReset)(pre(t)))),
                 crk_chan_reset_s:= crk_chan_reset_s3(
                      tim_c_key_s2(Abst_Key(m_ChanReset(t)),
                      tim_c_key_s(sdd3(m_ChanReset)(pre(t)))),
                      Abst_Key(m_ChanReset(t)),
                      Abst_pre_key(m_ChanReset)(pre(t)),
                      crk_chan_reset_s(sdd3(m_ChanReset)(pre(t))))#)
  ENDCOND
  MEASURE rank(t)

% Lemmas related SRS conditions to SDD conditions
 srs_sdd_keyon_heldfor_debounce:  LEMMA FORALL (t: noninit_elem):
   Held_For(LAMBDA (t:noninit_elem):e_KeyOn?(m_ChanReset(t)), k_Debounce)(t)
      IFF  keyon_e?(Abst_pre_key(m_ChanReset)(t)) &
           keyon_e?(Abst_pre_key(m_ChanReset)(pre(t)))

 srs_sdd_keyon_heldfor_stuck: LEMMA FORALL (t: noninit_elem):
   Held_For(LAMBDA t: e_KeyOn?(m_ChanReset(t)), k_Stuck)(t)
      IFF  (tim_c_key_s2(Abst_Key(m_ChanReset(t)),
            tim_c_key_s(sdd3(m_ChanReset)(pre(t)))) <= 1)

 srs_sdd_NotKeyon_heldfor_debounce: LEMMA FORALL (t: noninit_elem):
    Held_For(LAMBDA t: e_NotKeyOn?(m_ChanReset(t)), k_Debounce)(t)
      IFF  keyoff_e?(Abst_Key(m_ChanReset(t))) &
           keyoff_e?(Abst_Key(m_ChanReset(pre(t))))

% Abstract function: Abst_reset
Abst_reset_s(x: t_Reset): crk_reset_t =
    COND
    x = e_WaitForOn  -> wait_for_on_e,
    x = e_WaitForOff -> wait_for_off_e,
    x = e_ResetStuck -> reset_stuck_e,
    x = e_ClearTrip  -> clear_trip_e
    ENDCOND

% Special case when t=50
clr_eq_via_Sdd3_0_t50 : THEOREM
  Abst_reset_s(srs2(m_ChanReset)(50)) =
  crk_chan_reset_s(sdd3_0(m_ChanReset)(50))

clr_eq_via_Sdd3_t50 : THEOREM
  Abst_reset_s(srs2(m_ChanReset)(50)) =
  crk_chan_reset_s(sdd3(m_ChanReset)(50))

% Main block comparison theorem
srs_eq_sdd : THEOREM
  Abst_reset_s(srs2(m_ChanReset)(t)) =
  crk_chan_reset_s(sdd3(m_ChanReset)(t))
END ChanReset
```

F.2 The PVS proofs for the `ChanReset` theory

This subsection provides the proof scripts for the lemmas and theorems in the `ChanReset` theory. Among them, the lemmas `srs_sdd_keyon_heldfor_debounce`, `srs_sdd_keyon_heldfor_stuck`, and `srs_sdd_NotKeyon_heldfor_debounce` relate the *SRS conditions* to the *SDD conditions*. The theorem `srs_eq_sdd` is the main block comparison theorem.

Lemma `srs_sdd_keyon_heldfor_debounce`:

```
(""
 (SKOSIMP)
 (BDDSIMP)
 (("1"
   (EXPAND "Held_For")
   (EXPAND "heldfor")
   (GRIND)
   (EXPAND "heldfor")
   (GRIND))
  ("2"
   (EXPAND "Held_For")
   (EXPAND "heldfor")
   (BDDSIMP)
   (("1" (GRIND)) ("2" (GRIND))))
  ("3" (GRIND))))
```

Lemma `srs_sdd_keyon_heldfor_stuck`:

```
(""
 (SKOSIMP)
 (BDDSIMP)
 (("1"
   (EXPAND "tim_c_key_s2")
   (LIFT-IF)
   (BDDSIMP)
   (("1"
     (EXPAND "Held_For")
     (EXPAND "heldfor")
     (BDDSIMP)
     (("1" (GRIND))
      ("2"
       (EXPAND "sdd3")
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
     (("1" (GRIND))
      ("2"
       (GRIND)
       (EXPAND "sdd3")
       (GRIND)
       (("1" (EXPAND "heldfor") (GRIND))
        ("2"
         (EXPAND "sdd3" 2)
         (GRIND)
         (("1" (EXPAND "heldfor") (BDDSIMP) (EXPAND "heldfor") (GRIND))
          ("2" (EXPAND "heldfor") (EXPAND "heldfor") (GRIND))))
        ("3" (EXPAND "heldfor") (EXPAND "heldfor") (GRIND)))))))
  ("2" (GRIND))))
("2"
 (EXPAND "tim_c_key_s2")
 (LIFT-IF)
 (BDDSIMP)
 (("1"
   (TYPEPRED "t!1")
   (EXPAND "sdd3")
   (LIFT-IF)
   (ASSERT)
   (BDDSIMP)
   (("1" (HIDE 2 -1 -2 4) (GRIND))
    ("2"
     (EXPAND "tim_c_key_s2")
     (LIFT-IF)
     (BDDSIMP)
     (("1"
       (EXPAND "sdd3")
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
       (("1" (HIDE -1 -2 -4 -5 1 2 3) (GRIND))
        ("2"
         (EXPAND "tim_c_key_s2")
         (LIFT-IF)
         (ASSERT)
         (BDDSIMP)
```

```
           (("1"
             (EXPAND "sdd3")
             (LIFT-IF)
             (ASSERT)
             (BDDSIMP)
             (("1" (GRIND))
              ("2"
               (EXPAND "tim_c_key_s2")
               (LIFT-IF)
               (ASSERT)
               (BDDSIMP)
               (("1"
                 (EXPAND "Held_For")
                 (EXPAND "heldfor")
                 (BDDSIMP)
                 (("1" (GRIND)) ("2" (GRIND))))
                ("2" (GRIND))))))
            ("2" (GRIND))))))
        ("2" (GRIND))))))
    ("2" (GRIND))))))
```

Lemma `srs_sdd_NotKeyon_heldfor_debounce`:

```
(""
 (SKOSIMP)
 (BDDSIMP)
 (("1" (EXPAND "Held_For") (EXPAND "heldfor") (GRIND))
  ("2" (EXPAND "Held_For") (EXPAND "heldfor") (GRIND)) ("3" (GRIND))))
```

Theorem `srs_eq_sdd`:

```
(""
 (INDUCT "t" 1 "clock_induction")
 (("1" (SKOSIMP) (GRIND))
  ("2"
   (SKOSIMP)
   (SKOSIMP)
   (INST -1 "m_ChanReset!1")
   (EXPAND "Abst_reset_s")
```

```
(LIFT-IF)
(ASSERT)
(GROUND)
(("1"
  (EXPAND "srs2" -1)
  (EXPAND "f_ChanReset2")
  (LIFT-IF)
  (ASSERT)
  (BDDSIMP)
  (("1"
    (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
    (INST -1 "m_ChanReset!1" "t!1")
    (HIDE 1)
    (GRIND))
   ("2"
    (CASE "NOT Held_For(LAMBDA t:
       e_NotKeyOn?(m_ChanReset!1(t)), k_Debounce)(t!1)")
    (("1"
      (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
      (LEMMA "srs_sdd_keyon_heldfor_debounce")
      (INST -1 "m_ChanReset!1" "t!1")
      (INST -2 "m_ChanReset!1" "t!1")
      (GRIND)
      (("1" (HIDE -1 -3 -4 1 3 4 5) (EXPAND "sdd3") (GRIND))
       ("2" (HIDE -2 1 3 4 5) (EXPAND "sdd3") (GRIND))))
     ("2"
      (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
      (INST -1 "m_ChanReset!1" "t!1")
      (GRIND))
     ("3" (GRIND))))))
 ("2"
  (HIDE 2)
  (EXPAND "srs2" -1)
  (EXPAND "f_ChanReset2")
  (LIFT-IF)
  (ASSERT)
  (BDDSIMP)
  (LEMMA "srs_sdd_keyon_heldfor_debounce")
  (LEMMA "srs_sdd_keyon_heldfor_stuck")
```

```
     (INST -1 "m_ChanReset!1" "t!1")
     (INST -2 "m_ChanReset!1" "t!1")
     (BDDSIMP)
     (("1"
       (HIDE -1 -4 -5 1 3)
       (EXPAND "sdd3" 2)
       (EXPAND "crk_chan_reset_s3" 2)
       (LIFT-IF)
       (ASSERT)
       (GRIND))
      ("2"
       (HIDE -2 -3 1 2 4 5 6)
       (GRIND)
       (EXPAND "heldfor")
       (GRIND)
       (EXPAND "heldfor")
       (GRIND))
      ("3"
       (HIDE -2 -3 1 2 4 5 6)
       (GRIND)
       (EXPAND "heldfor")
       (GRIND)
       (EXPAND "heldfor")
       (GRIND))))
    ("3"
     (HIDE 2 3)
     (CASE "Held_For(LAMBDA t: e_KeyOn?(m_ChanReset!1(t)), k_Stuck)(t!1)")
     (("1"
       (LEMMA "srs_sdd_keyon_heldfor_stuck")
       (HIDE -3 -4 -5)
       (INST -1 "m_ChanReset!1" "t!1")
       (GRIND))
      ("2"
       (HIDE -3 2)
       (EXPAND "srs2")
       (LIFT-IF)
       (EXPAND "f_ChanReset2")
       (LIFT-IF)
       (ASSERT))
```

```
  ("3" (GRIND))))
("4"
 (HIDE -2 2)
 (CASE "srs2(m_ChanReset!1)(t!1) = e_ClearTrip")
 (("1"
   (HIDE 1 2 3)
   (EXPAND "srs2" -1)
   (EXPAND "f_ChanReset2")
   (LIFT-IF)
   (ASSERT))
  ("2" (GRIND))))
("5" (GRIND))
("6"
 (HIDE 2 3)
 (EXPAND "srs2" -1)
 (EXPAND "f_ChanReset2")
 (LIFT-IF)
 (ASSERT)
 (BDDSIMP)
 (("1"
   (LEMMA "srs_sdd_keyon_heldfor_debounce")
   (INST -1 "m_ChanReset!1" "t!1")
   (BDDSIMP)
   (("1"
     (HIDE -1 -4 -5 1)
     (EXPAND "sdd3" 1)
     (EXPAND "crk_chan_reset_s3" 1)
     (LIFT-IF)
     (ASSERT)
     (BDDSIMP)
     (REVEAL 1)
     (LEMMA "srs_sdd_keyon_heldfor_stuck")
     (INST -1 "m_ChanReset!1" "t!1")
     (BDDSIMP)
     (PROPAX))
    ("2"
     (HIDE -2 -3 2 3 4)
     (GRIND)
     (EXPAND "heldfor")
```

```
      (GRIND)
      (EXPAND "heldfor")
      (GRIND))
     ("3"
      (HIDE -2 -3 2 3 4)
      (GRIND)
      (EXPAND "heldfor")
      (GRIND)
      (EXPAND "heldfor")
      (GRIND))))
   ("2"
    (LEMMA "srs_sdd_keyon_heldfor_stuck")
    (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
    (INST -1 "m_ChanReset!1" "t!1")
    (INST -2 "m_ChanReset!1" "t!1")
    (ASSERT)
    (BDDSIMP)
    (("1"
      (HIDE -1 3 4)
      (EXPAND "sdd3" 3)
      (EXPAND "crk_chan_reset_s3")
      (LIFT-IF)
      (ASSERT)
      (BDDSIMP)
      (HIDE -1 2 3)
      (GRIND))
     ("2"
      (HIDE -1 3 4)
      (EXPAND "sdd3" 3)
      (EXPAND "crk_chan_reset_s3")
      (LIFT-IF)
      (ASSERT))))))
 ("7"
  (EXPAND "srs2" -1)
  (EXPAND "f_ChanReset2")
  (LIFT-IF)
  (ASSERT)
  (BDDSIMP)
  (LEMMA "srs_sdd_keyon_heldfor_stuck")
```

```
     (INST -1 "m_ChanReset!1" "t!1")
     (BDDSIMP)
     (HIDE -1 -3 2 3)
     (EXPAND "sdd3" 1)
     (EXPAND "crk_chan_reset_s3")
     (LIFT-IF)
     (ASSERT))
    ("8"
     (CASE "srs2(m_ChanReset!1)(t!1) = e_ClearTrip")
     (("1"
       (HIDE 1 3 4 5)
       (EXPAND "srs2" -1)
       (EXPAND "f_ChanReset2")
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
       (HIDE 1 2)
       (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
       (INST -1 "m_ChanReset!1" "t!1")
       (BDDSIMP)
       (HIDE -1 -4)
       (EXPAND "sdd3" 1)
       (EXPAND "crk_chan_reset_s3")
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
       (("1" (HIDE 1 -1 -4) (EXPAND "tim_c_key_s2") (GRIND)) ("2" (GRIND))))
      ("2" (GRIND))))
    ("9"
     (HIDE 2 3)
     (EXPAND "srs2" -1)
     (EXPAND "f_ChanReset2")
     (LIFT-IF)
     (ASSERT)
     (BDDSIMP)
     (HIDE 1 2)
     (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
     (INST -1 "m_ChanReset!1" "t!1")
     (BDDSIMP)
```

```
  (HIDE -1 -4)
  (EXPAND "sdd3" 1)
  (EXPAND "crk_chan_reset_s3")
  (LIFT-IF)
  (ASSERT)
  (BDDSIMP)
  (("1" (HIDE 1 -1 -4) (EXPAND "tim_c_key_s2") (GRIND)) ("2" (GRIND))))
 ("10"
  (HIDE 2 3 4)
  (EXPAND "srs2" -1)
  (EXPAND "f_ChanReset2")
  (LIFT-IF)
  (GRIND))
 ("11" (HIDE 2 3 4 5) (GRIND)) ("12" (GRIND))
 ("13"
  (CASE "srs2(m_ChanReset!1)(pre(t!1)) = e_ClearTrip")
  (("1"
    (HIDE 2 3 4)
    (EXPAND "srs2" -2)
    (EXPAND "f_ChanReset2")
    (LIFT-IF)
    (ASSERT)
    (BDDSIMP)
    (("1"
      (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
      (INST -1 "m_ChanReset!1" "t!1")
      (BDDSIMP)
      (HIDE -1 -4 1)
      (EXPAND "sdd3" 1)
      (EXPAND "crk_chan_reset_s3" 1)
      (LIFT-IF)
      (ASSERT)
      (GRIND))
     ("2"
      (CASE "Held_For(LAMBDA (t: clock[50]):
            e_NotKeyOn?(m_ChanReset!1(t)), k_Debounce)(t!1)")
      (("1" (GRIND))
       ("2"
        (LEMMA "srs_sdd_NotKeyon_heldfor_debounce")
```

```
           (LEMMA "srs_sdd_keyon_heldfor_debounce")
           (INST -1 "m_ChanReset!1" "t!1")
           (INST -2 "m_ChanReset!1" "t!1")
           (BDDSIMP)
           (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND))
            ("4"
             (HIDE 1 -1 3 5 6)
             (EXPAND "sdd3" 3)
             (EXPAND "crk_chan_reset_s3")
             (LIFT-IF)
             (ASSERT)
             (GRIND)
             (("1" (EXPAND "sdd3" -2) (GRIND))
              ("2" (EXPAND "sdd3" 2) (GRIND))))
            ("5" (GRIND)) ("6" (GRIND))))
          ("3" (GRIND))))))
      ("2" (GRIND))))
    ("14"
     (CASE "srs2(m_ChanReset!1)(pre(t!1)) = e_ClearTrip")
     (("1"
       (HIDE 3 4 5)
       (EXPAND "srs2" -2)
       (EXPAND "f_ChanReset2")
       (LIFT-IF)
       (ASSERT)
       (BDDSIMP)
       (HIDE -3 3 4)
       (EXPAND "srs2")
       (GRIND)
       (EXPAND "heldfor")
       (GRIND))
      ("2" (ASSERT))))
    ("15"
     (HIDE 2 3)
     (CASE "srs2(m_ChanReset!1)(pre(t!1)) = e_ClearTrip")
     (("1"
       (HIDE 2 3 4)
       (EXPAND "srs2" -2)
       (EXPAND "f_ChanReset2")
      (LIFT-IF)
      (BDDSIMP)
      (("1"
        (LEMMA "srs_sdd_keyon_heldfor_stuck")
        (INST -1 "m_ChanReset!1" "t!1")
        (GRIND))
       ("2" (GRIND))
       ("3"
        (LEMM "srs_sdd_keyon_heldfor_debounce")
        (INST -1 "m_ChanReset!1" "t!1")
        (BDDSIMP)
        (("1" (HIDE -1 -5 -4 1) (GRIND))
         ("2"
          (HIDE -1 -3 2 3 4)
          (GRIND)
          (EXPAND "heldfor")
          (GRIND)
          (EXPAND "heldfor")
          (GRIND))
         ("3"
          (HIDE -1 -3 2 3 4)
          (GRIND)
          (EXPAND "heldfor")
          (GRIND)
          (EXPAND "heldfor")
          (GRIND))))
       ("4" (GRIND))))
     ("2" (GRIND))))
   ("16" (GRIND))))
 ("3" (ASSERT))))
```

Bibliography

[1] R. Abraham, “Evaluating Generalized Tabular Expressions in Software Documentation”, CRL Rep 346, Telecommunication Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ont., Jan. 1997

[2] P. Alexander, *Sequent Calculus*, `http://www.ececs.uc.edu/~alex/teach/ece793/sequent.html`

[3] 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`.

[4] B. Dutertre and V. Stavridou, “Formal requirements analysis of an avionics control system,” *IEEE Transactions on Software Engineering*, vol. 23, pp. 267–278, May 1997.

[5] 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.

[6] 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*.

[7] 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.

[8] R. Janicki, D. L. Parnas, and J. Zucker, “Tabular representations in relational documents,” in *Relational Methods in Computer Science* (C. Brink, W. Kahl, and

G. Schmidt, eds.), Advances in Computing Science, ch. 12, pp. 184–196, Springer Wien NewYork, 1997.

[9] R. Janicki and R. Khédri, "On a formal semantics of tabular expressions," *Science of Computer Programming*, Vol. 39, pp. 189 – 213, 2001.

[10] M. Jing "A Table Checking Tool", SERG Rep 384, Software Engineering Research Group, McMaster University, March 2000.

[11] P. Joannou *et al.*, "Standard for Software Engineering of Safety Critical Software." CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1, Jan. 1995.

[12] M. Lawford and P. Froebel, "Application of tabular methods to the specification and verification of a nuclear reactor shutdown system." Submitted to Formal Methods in System Design.

[13] M. Lawford and H. Wu, "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, 2000.

[14] J. McDougall, M. Viola, and G. Moum, 'Tabular Representation of Mathematical Functions for the Specification and Verification of Safety Critical Software'. In: *SAFECOMP'94: The 13th International Conference on Compute Safety, Reliability and Security.* Anaheim, California, pp. 21–30, 1994.

[15] S. Owre, J. Rushby, and N. Shankar, "Integration in PVS: Tables, types, and model checking," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)* (E. Brinksma, ed.), vol. 1217 of *Lecture Notes in Computer Science*, (Enschede, The Netherlands), pp. 366–383, Springer-Verlag, Apr. 1997.

[16] S. Owre and N. Shankar and J. M. Rushby and D. W. J. Stringer-Calvert, *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[17] D. Parnas and J. Madey, "Functional documentation for computer systems engineering," Tech. Rep. CRL No. 273, Telecommunications Research Institute of Ontario, McMaster University, Sept. 1991.

[18] D. Parnas, "Tabular representation of relations" Tech. Rep. 260, Communications Research Laboratory, McMaster University, Oct. 1992.

[19] D. Peters, "Deriving Real-Time Monitors from System Requirements Documentation", SERG Report 383, Software Engineering Research Group, McMaster University, Jan. 2000.

[20] J. Rushby, S. Owre, and N. Shankar: 1998, 'Subtypes for Specifications: Predicate Subtyping in PVS'. *IEEE Transactions on Software Engineering* **24**(9), 709–720.

[21] J. Skakkebaek and N. Shankar, "Towards a Duration Calculus Proof Assistant in PVS", Springer-Verlag Lecture Notes in Computer Science Vol. 863, pp.660-679, 1994

[22] N. Shankar and S. Owre and J. M. Rushby and D. W. J. Stringer-Calvert, *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[23] N. Shankar, S. Owre, and J. M. Rushby, *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[24] N. Shankar, S. Owre, and J. M. Rushby, *Analyzing Tabular and State-Transition Requirements Specifications in PVS*. Computer Science Laboratory, SRI International, Menlo Park, CA, April 1996.

[25] F. Tian "Structured Decision Table $\Leftrightarrow$ Generalized Decision Table Conversion Tool", SERG Rep 380, Software Engineering Research Group, McMaster University, Sep. 1999.