Inspection of Concurrent Systems: Combining Tables, Theorem Proving and Model Checking

By
Vera Pantelic, B. Eng.

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

M.A.Sc.

Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Inspection of Concurrent Systems:
Combining Tables, Theorem Proving and Model Checking

AUTHOR: Vera Pantelic, B. Eng.(Belgrade University, Serbia & Montenegro)

SUPERVISOR: Dr. Mark Lawford & Dr. David Parnas

NUMBER OF PAGES: ix, 99

Abstract

A process for rigorous inspection of concurrent systems using tabular specification was developed and applied to the classic Readers/Writers concurrent program by Jin in [15]. The process involved rewriting the program into a table and then performing a manual "column-by-column" inspection for safety and clean completion properties. The key element in the process is obtaining an invariant strong enough to prove the properties of interest. This thesis presents partial automation of the proposed approach by combining theorem proving and model checking. Model checking is first used to validate a formal model of the system with a small, fixed number of concurrent process instances. The verification of the system for an arbitrary number of processes is then performed using theorem proving together with model checking on the earlier model to quickly validate potential invariants before they are used in the formal proof. This method was used to check the manual proof of the Readers/Writers problem given in [15], discovering several random and one systematic mistake of the proof. Then, a new, significantly automated proof was performed.

Acknowledgments

I would like to express my deep gratitude to my supervisors, Dr. Mark Lawford and Dr. David Parnas, for their guidance and help. Thanks to the committee members, Dr. Sanzheng Qiao and Dr. Ryszard Janicki, for their useful comments.

I would also like to thank Leonardo de Moura of SRI for his advice on using SAL, which helped shape the modeling and analysis in SAL. Thanks to Cesar Munoz for his fast responses to my queries about PVS.

Last, but not least, I thank my family and friends for their support. Special gratitude I owe to my mother. This thesis is dedicated to her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Inspection of concurrent programs still presents a challenge for software developers. The atomic actions of the processes constituting a concurrent program can be interleaved in many different ways. Furthermore, the concurrent software systems often lack the regularity of hardware systems. Thus, the nature of concurrent systems can make their state spaces large and irregular, making it extremely hard to ensure that all the possible behaviors of the system have been analyzed.

A reliable and effective inspection approach for the inspection of concurrent programs is proposed in [15]. Inspection is made easier and reliable by inspecting each of the components separately. Further, each component's behavior is described using program function tables [28]. However, as will be shown in this thesis, the manual proof of the correctness criterion given in [15] failed to explore the whole transition relation described by the program function table. Automated tool support, on the other hand, helped discover the flaws of the manual proof easily and was invaluable for properly proving both safety and liveness properties.

1.2 Our Approach

There are many different approaches to mechanized formal analysis of concurrent systems represented with transition relations. Those include deduction (theorem

proving), model checking, abstraction and model checking, automated abstraction, bounded model checking [30, 11], and equivalence verification [19, 21, 20].

Model checking is a technique for verifying finite state concurrent systems [2]. First, a model of the program is to be built. Next, the properties of the system are specified, usually in temporal logic. If the model fails to satisfy the property, a counterexample is produced that demonstrates a behavior that satisfies the negation of the property. The most important advantage of model checking over theorem proving is that it is completely automatic. However, although the state explosion problem has been addressed by many techniques (e.g., partial order reduction, infinite-state model checking), model checking still cannot handle systems with an arbitrarily large number of processes.

Deductive verification (theorem proving), on the other hand, can be used to analyze very large or infinite systems. It still remains the most general way to reason about complex systems. However, it can be a tedious and time-consuming process that requires substantial human guidance.

This thesis represents an extension of the approach of [15], providing partial automaton of the proposed inspection process. The original program can be analyzed in SPIN. SPIN is a model checking tool specialized for handling concurrent systems. Its specification language provides the primitives for interprocess communication [14]. Model checking in SPIN can be particularly useful for purpose of refutation (generating a counterexample for a particular version of the system). Full verification, however, requires the use of theorem proving, since the number of the processes can be arbitrarily large, and the values of global or local process variables can be unbounded.

The starting point of the full verification is the program function table prepared as in [15]. The transition relation of the concurrent system as given by the table is rewritten into the SAL model checker and model checked for safety and liveness properties. However, at this point, SAL supports neither tables, nor does it offer a full typechecker. The table is then rewritten into the PVS specification language table construct and checked for consistency and completeness. Safety properties are proved in PVS using the inductive invariant approach [30]. The property P is inductive on transition relation T and set of initial states I if it includes all the initial states ($I(s) \Rightarrow P(s)$) and is closed on all the transitions ($P(s) \wedge T(s,t) \Rightarrow P(t)$). We try to prove that a safety property is an invariant of the system, by showing that it is satisfied in the

initial state and preserved by any transition of the system. However, few properties are inductive. Failed goals indicate the auxiliary invariants that we then use to strengthen the initial property. Then, we try to prove that the strengthened invariant (conjunction of the newly found ones and the desired invariant) is inductive. Before being checked in theorem prover, every new, auxiliary invariant is model checked in the SAL model-checker for a specific instance of the problem. This check is automatic and fast. The process iterates until the inductive invariant is found or it is suggested by the failed proof(s) that a proof of inductivity cannot be found. Proving liveness property then requires the additional strengthening of the found inductive invariant.

1.3 Contribution of the Thesis

We believe that the contributions of this work are:

1. We provided partial automation of the inspection process of [15].
2. We illustrated the necessity of the computer-aided verification of the concurrent systems in inspection of [15] by automating the manual proof of the safety property of the Readers/Writers problem (as in [15]). Not only were we able to significantly reduce the effort needed to complete the proof (the manual proof of the safety property is 100 pages long), but we also discovered several inadvertent and one systematic mistake in the manual proof. We managed to automate the proof of the safety property almost completely using PVS strategies.
3. Theorem proving and model checking were successfully combined. Two model checking tools (one of which is specialized for models of concurrency, the other one with an input language very close that of the theorem prover) were used for model checking the classical concurrent program. Model checking potential invariants before using them in the theorem prover reduced the time required to obtain an inductive invariant compared to using only the theorem prover.
4. The thesis provides a detailed example of the computer-aided verification of a concurrent programs with an arbitrarily large number of processes.

Model checking tools were used for refutation purposes - for finding the bugs in both the original program and the one rewritten into table. Moreover, SAL was used

for checking the auxiliary invariants found in PVS. PVS provided almost complete automation of the consistency and coverage checks of the tabular specification. Failed goals generated in PVS indicated the auxiliary invariants. The proof was automated using PVS strategies. The PVS user strategies are given in Appendix C.3, and PVS built-in strategies are given in [33].

1.4 Structure of The Thesis

- Chapter 2 represents an overview of the inspection of the concurrent programs with a detailed description of the inspection process of [15] applied to the classical concurrency problem, the Readers/Writers Problem [4].
- Chapter 3 provides an overview of the model-checking tools SPIN and SAL, and the PVS specification and verification system.
- Chapters 4 and 5 represent our approach applied to the Readers/Writers problem, formulated as in [15].
- Chapter 6 reports on the conclusions of this project and makes suggestions for future work.

Chapter 2

Inspection of Concurrent Programs

The material in this chapter is an important part of the background for the research presented in this thesis. It provides the reader with essential information on inspection of concurrent systems and inspection based on tables. Further, a detailed description of the inspection of concurrent systems using tables is given. This inspection approach and the example presented here form the basis of our research.

2.1 Formal Modeling of Concurrency

There are many different models of concurrency intended for the formal verification of concurrent systems. Petri nets represent one well-known formalism [23]. Axiomatic systems for concurrency are based on Hoare's logic [13] or Dijkstra's weakest precondition logic [8]. Extensions of those include the Lamport extension of Hoare logic [12], the Owicki-Gries extension of Hoare logic [24], and the Lamport extension of Dijkstra's weakest precondition logic [16].

A number of process algebras have been proposed. CCS (Calculus of Communicating Systems) and CSP (Communicating Sequential Processes) specify a concurrent systems as consisting of processes that are completely independent except for the communication between them [1]. CCS was developed as a formalism for describing multiprocess systems and exploring the notions of equivalence of processes [20]. CSP was initially developed as a programming language [12]. SCCS (Synchronous CCS) was developed to extend the CCS with the notion of synchronization between agents

[3]. However, the cost of applying the mentioned methods in software engineering has generally proven to be too high [1].

2.2 About Inspection Based on Tables

Tables are multi-dimensional mathematical expressions describing mathematical functions and relations. They were proposed in [28]. Tables have proven to be a useful method for software inspection, providing clarity in reading and understanding, and easiness in ensuring input domain coverage and consistency.

Tables were first used at the U.S. Naval Research Laboratory in the 1970s for the inspection process of the A-7E aircraft software [32]. Another inspection process based on tables was developed and applied in the Darlington Nuclear Power Generating Station and first reported in [29]. In [26] a rigorous inspection approach based on program-function tables was presented.

The application of tool-supported tabular methods to the specification and verification of safety-critical software for the Darlington Nuclear Power Generation Station was described in [17, 18].

The Display method, a method of documenting well-structured programs, is described in [27]. The application of the combination of this method and theorem proving in PVS was used in [31] for the inspection of the source code implementing the PPP protocol in Linux. We did not feel the need to use displays in this thesis, since the example program used is not a long one.

The details on the semantics of tables and type of tables used in this thesis are given in Section 3.3.

2.3 The Inspection of Concurrent Programs Using Tables

Note: The material presented in this section is taken mostly from [15].

2.3.1 Introduction to the Approach

In our model a concurrent program begins its execution from the initial state and advances while interleaving with other components. The key idea of this approach is the use of the "divide and conquer" principle: the correctness of the program components implies the correctness of the whole program.

The process includes the following:

1. Auxiliary variables are introduced to capture all the information needed to analyze the program.

2. The requirements of the program are formulated as a mathematical specification.

3. The primitive operators are specified (e.g., synchronization primitives) — this should have been done before the program was written.

4. The program is rewritten so that each primitive statement has a label. The transfer of control from statement to statement is made explicit by assigning a label value to an auxiliary variable (that functions as the program instruction counter) for each statement. The value of this auxiliary variable is the condition of the execution of each statement.

5. The program is described in a tabular representation.

6. Two properties of a concurrent program are to be proved:

   - Invariant property — ensures that the requirement predicate holds in all the reachable states of the program. A set of invariants that embodies the essential properties of the execution and is inductive is formulated.
   - Liveness property — ensures that all of the program's constituent processes can cleanly finish their execution.

The program is inspected to show that the invariant is satisfied in the initial state of the system and the execution of every primitive statement maintains the invariant, and that the liveness property holds.

2.3.2 Example Application: Readers/Writers Problem

One typical concurrency problem is the Readers/Writers problem [4]. Two different kinds of processes, readers and writers, access the common resource. An unlimited number of readers can concurrently access the resource, whereas a writer must have exclusive access to the resource. Among two variants of this problem presented in [4], the one that gives readers priority over the writers is chosen (the readers' preference is weak - if at least one reader is accessing the critical section, and both another reader and writer arrive, then the new reader gets preference over the writer. If, however, the writer leaves the critical section, and there are both readers and writers waiting to enter it, choice of which type of process is permitted to enter the critical section is arbitrary).

The Original Program

The program used to solve the chosen variant from [4] is reproduced below:

```
integer rdcnt; (initial value = 0)
semaphore mutex, w: (initial value for both = 1)
READER: P(mutex);                  WRITER:  P(w);
        rdcnt := rdcnt+1;                   WRITE;
        if rdcnt=1 then P(w);               V(w);
        V(mutex);
        READ;
        P(mutex);
        rdcnt := rdcnt-1;
        if rdcnt=0 then V(w);
        V(mutex);
```

Two semaphores are used as synchronization primitives. Semaphore w is used as a mutual exclusion semaphore for the first and the last reader, and any writer entering the critical section, while semaphore *mutex* ensures that only one reader process can enter or leave the critical section at a time. The variable `rdcnt` counts all the reader processes who have entered the critical section (meaning, the section protected with the `w` semaphore) or have asked for the permission to enter it.

Let rd and wt be the number of active reader and writer processes, respectively.

The informal requirement of the program as stated at the beginning of the subsection (at most one writer can write while no reader is reading, and any number of readers can read concurrently) can be written as the safety property:

$$(rd = 0 \vee wt = 0) \wedge wt < 2 \tag{2.1}$$

Applying the proposed approach to the example application

Applying the steps of the proposed approach (as described in the Section 2.3.1), the original Readers/Writers program can be rewritten as in Figure 2.1.

The *stop* symbol tells us when a process under execution can be interrupted, allowing other processes to resume their execution, i.e., each line of Figure 2.1 represents a primitive statement.

If more than one process is ready to execute, the choice of the process to be executed is non-deterministic. The array variable *next* functions as an instruction counter variable, locating the execution of each process — the value of $next[i]$ represents the current statement label of the i^{th} process. The labels $waitAtPm1$, $rlseAtPm1$, $waitAtPm2$, $rlseAtPm2$, $waitAtPwr$, $rlseAtPwr$, $waitAtPww$, $rlseAtPww$ are introduced so that synchronization primitives can be specified. A process can pass $P(sem)$ successfully (advance with its execution), it can be suspended (in which case it gets labeled as $waitAtPsem$), or released by a V-operation, in which case it acquires the label $rlseAtPsem$. The detailed specification of P/V operations of a semaphore is taken from [15] and reproduced in Appendix A.

The program is then rewritten into the table given in the Appendix B, originally taken from [15]. For these purposes, a parameter k $(0 < k \leq M)$ is introduced to denote the identification of a representative process. The pID represents the identification of the currently executing process. Two additional boolean expressions are introduced: $IsReader$ and $IsWriter$, that stand for $0 < k \leq n$ and $n < k \leq M$, respectively, where n is the number of reader processes and $0 \leq n \leq M$. The interested reader is referred to [15] for the details on rewriting the program as in Figure 2.1 to the table. The program state can be described as a 7-tuple $(rdcnt, rd, wt, mutex, w, next, pID)$.

```
READER i:
1 Begin
2 if next[i]=r1 then P(mutex) stop
3 if next[i]=waitAtPm1 then next[i]:= waitAtPm1 stop
4 if next[i]=rlseAtPm1 then next[i]:=r2 stop
5 if next[i]=r2 then rdcnt := rdcnt+1; next[i]:=r3 stop
6 if next[i]=r3 then if rdcnt=1 then P(w); rd := rd+1; stop
7 if next[i]=waitAtPwr then next[i]:=waitAtPwr stop
8 if next[i]=rlseAtPwr then rd := rd+1; next[i]:=r4 stop
9 if next[i]=r4 then V(mutex) stop
10 if next[i]=r5 then READ; next[i]:=r5 stop
11 if next[i]=r6 then P(mutex) stop
12 if next[i]=waitAtPm2 then next[i]:=waitAtPm2 stop
13 if next[i]=rlseAtPm2 then next[i]:=r7 stop
14 if next[i]=r7 then rdcnt := rdcnt-1; next[i]:=r8 stop
15 if next[i]=r8 then if rdcnt=0 then V(w); rd := rd-1 stop
16 if next[i]=r9 then V(mutex) stop
17 End

WRITER j:
1 Begin
2 if next[j]=w1 then P(w); wt := wt+1; stop
3 if next[j]=waitAtPww then next[j]:=waitAtPww stop
4 if next[j]=rlseAtPww then wt := wt+1; next[j]:=w2 stop
5 if next[j]=w2 then WRITE; next[j]:=w3 stop
6 if next[j]=w3 then V(w); wt := wt-1 stop
7 End
```

Figure 2.1: Readers/Writers program rewritten

Showing Clean Completion

We say that a program has a clean completion when all of its constituent processes can finish the execution (the program counter of every process can reach the label *EOP*). For the purposes of proving the clean completion of the program (liveness property), the vector of decreasing quantity DQ is defined in [15]:

$$DQ = (Pros, IntRW(next[1]), IntRW(next[2]), \ldots, IntRW(next[M]))$$

where M is the total number of processes, $Pros$ is the number of the processes that have not reached the *EOP* label yet, and $IntRW$ is the function mapping all the values of *next* to integers, as indicated in the Table 2.1.

x	$IntRW(x)$
$r1$	15
$waitAtPm1$	14
$rlseAtPm1$	13
$r2$	12
$r3$	11
$waitAtPwr$	10
$rlseAtPwr$	9
$r4$	8
$r5$	7
$r6$	6
$waitAtPm2$	5
$rlseAtPm2$	4
$r7$	3
$r8$	2
$r9$	1
$w1$	5
$waitAtPww$	4
$rlseAtPww$	3
$w2$	2
$w3$	1
EOP	0

Table 2.1: The $IntRW$ function definition

Let $l = 1, 2$. Suppose that, at the state l, the $next_l$ is the value of *next*, and $Pros_l$

is the number of processes (meaning, all the processes with a label assigned, except for those with the label *EOP*). As before, n is the number of the reader processes ($0 \leq n \leq M$). Let

$$\sum r_l = \begin{cases} 0, & n = 0 \\ \sum_{i=1}^{n} IntRw(next_l[i]), & 0 < n \leq M \end{cases} \tag{2.2}$$

$$\sum w_l = \begin{cases} 0, & n = M \\ \sum_{i=n+1}^{M} IntRw(next_l[i]), & 0 \leq n < M \end{cases} \tag{2.3}$$

$$DQ_l = (Pros_l, IntRW(next_l[i]), \ldots, IntRW(next_l[i]) \tag{2.4}$$

Then, the order property of DQ is given by the Table 2.2 where *DQorder* stands for $DQ_1 > DQ_2$.

	$Pros_1 > Pros_2$	$Pros_1 = Pros_2$		$Pros_1 < Pros_2$
		$\sum r_1 + \sum w_1 > \sum r_2 + \sum w_2$	$\sum r_1 + \sum w_1 \leq \sum r_2 + \sum w_2$	
DQorder	TRUE	TRUE	FALSE	FALSE

Table 2.2: The order property of DQ

Theorem of DQ 1 *Assume that there are no new readers/writers arriving. Then:*

1. *If there is a change of state other than a simple change of the pID variable, DQ decreases.*
2. *If there is no possible change of state other than a simple change of the pID variable, DQ is zero.*
3. *If DQ is zero, there is no waiting process.*

The decreasing quantity approach originates from the verification of the loops. The idea of proving the clean completion using this approach is to find an integer variable which, when initialized with positive value, will decrease if the program is making progress; if there is no progress possible, the value of the decreasing quantity variable should be zero, which in turn should mean that there is no waiting process. In this particular case, the vector DQ was chosen to be such a variable.

Chapter 3

Introduction to SPIN, SAL, and PVS

This chapter provides basic information on tools used for the research in this thesis: the model-checking tools SPIN [14] and SAL [22], and the theorem prover PVS [25].

3.1 The SPIN Model Checker

Note: Material presented in this section is a summary of [14].

SPIN is a tool for model-checking concurrent systems. Systems are modeled using a specification language called Promela (the name SPIN is an acronym for Simple Promela Interpreter). The language is targeted to the description of concurrent software systems, rather than the description of hardware circuits.

The basic building blocks of SPIN are asynchronous processes, buffered and unbuffered message channels, synchronizing statements, and structured data. There is no notion of time or clock; there are only a few computational functions and no floating point numbers. The emphasis of the language is on the synchronization and communication, not the computation.

SPIN is an "on-the-fly" model-checker: it does not precompute the entire global state graph as a prerequisite for the verification. Correctness properties can be specified as system or process invariants (using assertions), as LTL requirements, as Buchi

Automata, or as general omega-regular properties in the syntax of `never` claims. Some liveness properties can be verified only by compiling the model with the corresponding option.

SPIN can be used in two basic modes: as a simulator and as a verifier. As a simulator, it provides a means of random, guided and interactive simulations. As a verifier, it offers efficient checking of user specified requirements or validation of very large models with maximal coverage of the state space. The proof techniques it applies are based on either depth-first or breadth-first search, optimized with partial order reduction techniques and BDD-like storage techniques.

3.2 SAL

Note: The material presented in this section is mostly taken from [22], [6], and [5].

SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving and model checking towards the calculation of properties (symbolic analysis) of transition systems. The key part of the SAL framework is a language for describing transition systems. The language serves as a specification language and as the target for translators that extract the transition system description for popular programming languages such as Esterel and Java. The language also serves as a common source for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language.

The basic unit of specification in SAL is a module. Modules can be separately analyzed and composed synchronously or asynchronously. A module consists of a *state* type, an invariant definition on this state type, an initialization condition on this state type, and a binary *transition relation* on the state type. The state type is defined by four pairwise disjoint sets of *input*, *output*, *global*, and *local* variables. The transition rules are constraints on the current and next states of the transition, given either as guarded commands or as invariant definitions.

The current SAL toolset provides explicit state, symbolic, bounded, infinite bounded and witness model checkers for SAL. We will use the symbolic model checker called *sal-smc*, which uses linear temporal logic (LTL) as its assertion language. More-

over, properties can be specified in computation tree logic (CTL) if they are in the intersection of these two languages, in which case they are internally converted into LTL. However, the current version of SAL provides counterexamples only for LTL properties.

3.3 PVS

This section provides the background information on PVS. We review PVS capabilities, properties of the sequent calculus on which PVS is based, tabular specification and their support in PVS.

3.3.1 The PVS Language and Proof Checker

Note: The material presented in this subsection is largely based on [9].

PVS stands for "Prototype Verification System". It provides mechanized support for specification and verification: it offers a specification language in which mathematical theories and conjectures can be defined, and then, latter can be discharged using the interactive theorem prover. The specification language of PVS is based on higher-order logic, which is extended with predicate subtypes and dependent types, and a theory system. Its type constructors include functions, tuples, records, recursive datatypes (e.g., lists and trees), and enumerations; sets are represented by their characteristic predicates. A prelude of hundreds of theories contains many definitions, axioms and proved theorems; user-contributed libraries provide many additional theories.

The PVS theorem prover is interactive. It is based on a sequent calculus presentation. PVS offers the graphical representation of proofs in the form of proof trees. Proofs can be saved as scripts and rerun either automatically, or in a single-step mode. While basic proof commands are built-in, most are programmed as strategies. The built-in commands provide very powerful automaton that include decision procedures for ground (unquantified) integer and linear arithmetic, automatic rewriting, and BDD-based propositional simplification and symbolic model-checking.

Predicate subtypes offered by the PVS specification language allow for a great deal of specification to be embedded in its types, contributing clarity and economy

in specification. Since the predicate used for defining a predicate subtype can be arbitrary, typechecking can become undecidable, and may lead to proof obligations called type correctness conditions (TCCs). Typically, the proof strategies built into the theorem prover can automatically discharge some of these obligations; the harder ones are left for the user to guide the proof.

PVS in combination with SAL is chosen for the following reasons:

- PVS has a construct for tabular specification. The construct generates proof obligations to ensure that the column conditions are disjoint and complete.
- Since the table construct is highly integrated with the other capabilities of PVS, we were able to prove the invariant property and clean completion theorem without first converting the tabular expressions to equivalent logical expressions.
- Although PVS has a model checker integrated with its theorem prover, it lacks the counterexample generation capability and is not particularly fast.
- The specification language syntax of the model checking tool SAL is similar to that of PVS. Although automatic translators from one tool to another are not available yet, we found it easy to rewrite the SAL specification into a PVS specification.
- SAL is an open system intended for the integration and cooperation of different tools for symbolic analysis and will feature tighter integration with PVS in the future [9].

3.3.2 The Sequent Calculus of PVS

Note: The material presented up to the end of this chapter is mostly based on [35] and [17].

Let $P_i,\ i = 1, \ldots, n$ and $Q_j,\ j = 1, \ldots, m$ be formulas in higher order logic and $\vdash$ is used to denote a syntactic entailment. Now, $\neg P_1,\ P_1 \wedge Q_1,\ P_1 \vee Q_1$ and $P_1 \Rightarrow Q_1$ denote negation, conjunction, disjunction and implication respectively. In general, assuming that the properties of the system inputs are all true (the P_i's), we want to prove that at least one of the output properties (one or more of the Q_i's)

is true. We formally write, $P_1, P_2, \ldots, P_n \vdash Q_1 \vee Q_2 \vee \ldots \vee Q_m$, or equivalently, $P_1 \wedge P_2 \wedge \ldots \wedge P_n \vdash Q_1 \vee Q_2 \vee \ldots \vee Q_m$. This expression is called a sequent. If the proof for it can be found, the sequent is valid. In sequent calculus this is written as in Figure 3.1.

$$\frac{P_1, P_2, \ldots, P_n}{Q_1 \vee Q_2 \vee \ldots \vee Q_m} \qquad \text{or} \qquad \begin{array}{l} P_1 \\ P_2 \\ \vdots \\ P_n \\ \hline Q_1 \\ Q_2 \\ \vdots \\ Q_n \end{array}$$

Figure 3.1: Sequents in sequent calculus

Proofs are done by transforming the sequent into one of these forms:

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

Here $\top$ and $\bot$ denote TRUE and FALSE, respectively.

3.3.3 Tabular Specification of Functions

The function $f : T_1 \times T_2 \times \ldots \times T_m \to T_r$ has the following tabular representation:

$$f(x_1, \ldots, x_m) = \begin{array}{|c|c|c|c|} \hline c_1 & c_2 & \ldots & c_n \\ \hline e_1 & e_2 & \ldots & e_n \\ \hline \end{array} \quad \text{or} \quad \begin{array}{|c|c|} \hline c_1 & e_1 \\ \hline c_2 & e_2 \\ \hline \ldots & \ldots \\ \hline c_n & e_n \\ \hline \end{array} \tag{3.1}$$

where each c_i is a predicate and e_i is a term of type T_r. The interpretation is that when a given condition c_i is true, f is equal to e_i. For the table to properly define a (total) function, two conditions should be satisfied:

1. Disjointness requires that each distinct pair of conditions c_i, c_j is disjoint, i.e., $i \neq j \Rightarrow \neg(c_i \wedge c_j)$.

2. Completeness requires that the disjunction of all the c_i's is true, i.e., $(c_1 \vee c_2 \vee \ldots \vee c_n)$ evaluates to $TRUE$.

Therefore, for a given $x_1, \ldots, x_m$ only one c_i can be true.

Consider the example, $sign(x)$, for $x \in \mathbb{R}$:

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

which can be specified as a table:

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

3.3.4 The PVS $COND$ Construct

For specification by cases the standard PVS language offers $COND$ construct, as indicated on the left side of Figure 3.2.

The right side of Figure 3.2 shows the equivalent IF-THEN-ELSE statements that PVS uses as the internal interpretation of the COND statement. While much of the typechecking required to ensure conservative extension of PVS logic can be done automatically, predicate subtypes (as mentioned earlier) and tabular specification of functions can cause PVS to generate TCCs. Use of COND causes PVS to automatically generate Disjointness and Completeness TCCs. These are often automatically proved by built-in proof strategies. In case these strategies fail, the resulting unprovable sequents can often provide useful information regarding the incompleteness or inconsistency of specifications.

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

Figure 3.2: COND construct and PVS interpretation

The following is the PVS definition of $sign(x)$ function using the PVS COND construct:

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

Typechecking the previous segment generates the following TCCs, which are automatically discharged.

```
 % Disjointness TCC generated (at line 11, column 1) for
  % COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
sign_cond_TCC1: OBLIGATION
  FORALL (x: real):
    NOT (x < 0 AND x = 0) AND
     NOT (x < 0 AND x > 0) AND NOT (x = 0 AND x > 0);

% Coverage TCC generated (at line 11, column 1) for
  % COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
sign_cond_TCC2: OBLIGATION FORALL (x: real): x < 0 OR x = 0 OR x > 0;
```

3.3.5 The PVS *TABLE* Construct

PVS has various TABLE constructs that provide more readable prover input. They are internally translated to PVS COND constructs for typechecking and proving purposes. Consider the table in Figure 3.3.

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

Figure 3.3: One-dimensional vertical table in PVS

Horizontal lines in Figure 3.3 are simply comments. This specification is equivalent to that of `sign_cond`, it generates the same TCCs and is treated the same as the equivalent `IF-THEN-ELSE` in the proofs. In this thesis we will use only one-dimensional vertical tables. For detailed information on PVS' support for other types of tables (enumeration tables, data type tables, one-dimensional horizontal and two-dimensional tables), the interested reader is referred to [35].

Chapter 4

Model Checking The Readers/Writers Problem

In this chapter we show how the original version of the Readers/Writers concurrent program with a fixed number of readers and writers can be formalized and model-checked. We use the SPIN model checker (since it is specialized for concurrent programs) for refutation purposes: some potential bugs of the program can be discovered in this early stage of the verification. Then, we formalize the program, rewritten as a tabular specification, to match the SAL specification language, in order to model check it for safety and liveness properties. This not only allows potential bugs of the original program to be discovered, but also the potential errors in the rewritten specification. We will use the SAL model as a prelude to theorem proving of the general model with an arbitrary number of readers and writers (as will be shown in the next chapter): every potential auxiliary invariant found by PVS is model checked in SAL.

4.1 Model Checking The Original Version In SPIN

This section first presents the modeling of the original Readers/Writers program in PROMELA, the specification language of SPIN. Then, the analysis of this model is performed using the SPIN model checker.

4.1.1 Specification in SPIN

SPIN supports rendezvous and buffered message passing, and communication through shared memory.

The semaphores used for synchronization in the Reader/Writer problem are easily modeled as shown in Figure 4.1. Semaphore `mutex`, which ensures that only one reader

```
mtype {p, v};
chan mutex = [0] of {mtype};
active proctype m1()
{
        byte count=1;
        do
        :: (count == 1) ->
 end:           mutex!p; count = 0
        :: (count == 0) ->
                mutex?v; count = 1
        od
}
```

Figure 4.1: Semaphore in SPIN

will enter or leave the critical section at the time, is modeled by the process of type `m1` with the help of the rendezvous port `mutex`. (The semaphore `w`, the mutual exclusion semaphore for the first and the last reader, is modeled in the same way.) A rendezvous port is a channel of capacity zero, that can only pass, but cannot store messages [14]. Message interactions via such rendezvous ports are, by definition, synchronous. The syntax for specifying a message transmission is borrowed from Hoare's CSP language: the `send` operator is represented with an exclamation mark and the `receive` operator is represented by a question mark. The label `end` will be explained later.

The definition and instantiation of the writer processes (two of them) are given in Figure 4.2. The label `eopw` will be explained later.

Compared to the original program, our SPIN model contains the additional global variables `rd` and `wt` (as in [15]), whose values are updated as a part of the same `atomic` sequence in which a process enters/leaves the critical section. The variables `rd` and `wt` are used as the counters of all the active readers and writers, respectively, in the

```
active [2] proctype writer()
{
          atomic{
                w?p; wt++
                };
          skip;
          atomic{
                 wt--; w!v
                };
}
```

Figure 4.2: Modeling writer processes in SPIN

read/write section. The complete SPIN code is given in Appendix C.1.

4.1.2 Analysis in SPIN

Safety Property: The safety property defined as

$$(rd = 0 \vee wt = 0) \wedge wt < 2 \wedge rd \geq 0 \wedge wt \geq 0 \tag{4.1}$$

can be checked in SPIN using a `never` claim. We note that the safety property as given here is a modified version of the property defined in Equation 2.1 (originally taken from [15]). Since the *rd* and *wt* variables are integers, adding the last two conjuncts as in equation 4.1 requires that number of readers/writers cannot be negative). We use a `never` claim to specify the behavior that should never happen, i.e., it is never the case that equation 4.1 is false:

```
never
{
do
:: !((rd == 0 || wt == 0) && wt < 2 && rd >= 0 && wt >= 0) -> break
:: else
od
}
```

The check can be done for the model in which processes repeatedly execute the piece of code (do not terminate).

Liveness property: The liveness property defined in Section 2.3.1 requires that every path of the system will eventually reach the state where all the reader/writer processes have reached the end of their execution. This check can be done in SPIN by checking for the absence of the invalid end states. By default, the only valid end states in SPIN are those in which every process that was instantiated has reached the "end" of its code. We used the labels `end` in the `m1` and `m2` processes so that a state in which all the readers/writers have finished the execution would not be flagged as an invalid one. So, without the `end` labels, in checking our model for invalid end states, a state with all the readers/writers at the end of their execution would be marked as an invalid one. In verification mode, SPIN checks for the invalid end states by default.

The SPIN model checking results are given in Table 4.1. All the computations as presented in this thesis were performed on a dual 2.4 GHz Xeon machine with 4 GB of RAM running RedHat Linux 9.0.

	safety/completion	
	states	time(s)
3R/2W	3619	0.02
5R/5W	$0.4 \cdot 10^6$	1.25
6R/6W	$2.3 \cdot 10^6$	115
8R/8W	$8.4 \cdot 10^7$	6555
10R/10W	-	>20h

Table 4.1: SPIN model checking results

From Table 4.1 it is obvious that checking the properties even for the system of 8 readers and 8 writers is very slow. We can use the SPIN's approximation techniques described in [14] (collapse compression, bitstate hashing, hash-compact) to make a quick check, but these techniques do not guarantee the complete coverage, and are, therefore, used only as a last resort. Moreover, even if the size of the state space would be manageable, the maximal number of processes allowed in a PROMELA model is 255.

4.2 Formalization of Readers/Writers Problem in SAL

In this section, the Readers/Writers problem is rewritten to match the table from Appendix B. SAL does not support tables, so the table is rewritten into the transition part of the SAL module: table headers are rewritten into the guards, and cells into the assignment part of the guarded commands. Safety and liveness properties are model checked using SAL's symbolic model checker for refutation purposes since some bugs might have been introduced while rewriting the program into tabular specification. Then, the SAL model will be used for checking the auxiliary invariants found in PVS.

4.2.1 Specification in SAL

Figure 4.3 contains a part of the context `rw` with type declarations. The context `rw` has two parameters: the number of processes `M`, and the number of reader processes `n`. The system state is of record type `state`, which consists of the fields `m`, `w`, `rdcnt`, `next`, `rd`, and `wt`. The fields `m` and `w` are of the `sem` record type. This type consists of the `cnt` and `set` fields. The field `m` functions as a mutual exclusion semaphore for readers to ensure that only one reader will enter or leave the critical section at a time. The field `w` provides mutual exclusion in the critical section shared by both readers and writers. The field `rdcnt` counts all the readers that have entered or are still waiting to enter the critical section. The elements of the array `next` are used to store the process states by specifying a process's next executing statement (as explained in Section 2.3.2). These elements are of type `label`.

Since we are using SAL's symbolic model checker for finite state systems, the types of the fields of the global state cannot be unbounded. That is why we needed the subrange type `semtype` as the type of field `cnt` of type `sem`, fields `rd` and `rdcnt` of type `rdtype`, and `wt` of type `wttype`. The types are given with the tightest bounds possible, in order to minimize the number of BDD variables (model checking is faster), but also to enable the check that the variables of these types never go over the bounds (see the `typecheck2` theorem in the next section). Users perform this typecheck because the full typechecker for SAL is not available yet; the present one does not detect overflows.

```
rw{; M : nznat, n : nat}: CONTEXT =
BEGIN
  Job_Idx: TYPE = [1..M];
  label: TYPE = {r1, waitAtPm1, rlseAtPm1, r2, r3, waitAtPwr, rlseAtPwr,
                 r4, r5, r6, waitAtPm2, rlseAtPm2, r7, r8, r9, w1, w2, w3,
                 waitAtPww, rlseAtPww, EOP};
  rdtype: TYPE = [-1..n+1];
  wttype: TYPE = [-1..(M-n+1)];
  semtype: TYPE = [-M..2];
  index: TYPE = [1..M];
  sem: TYPE = [#cnt: semtype,
                set: setof #];
  state: TYPE = [#
                m: sem,
                w: sem,
                rdcnt: rdtype,
                next: ARRAY index OF label,
                rd: rdtype,
                wt: wttype #]
            .
            .
            .
```

Figure 4.3: The context `rw`

Referring to the SAL input files in Appendix B, the parametric module `process` is used to specify the behavior of a reader/writer process. We could have defined two different parametric modules, one for readers, and one for writers. Instead, we decided to use only one, so that the state machine it models more closely resembles the original function table from [15] and more direct comparison to the manual proof from [15] can be made. The process local bool variable `IsReader` is initialized with TRUE if the $pID \leq n$, and FALSE otherwise.

The transition relation is described in the `TRANSITION` part of the module. The guard commands of the transition relation are labeled by the number of the column they refer to in the Figure B.1 in Appendix B, originally taken from [15]. There is no built-in support in SAL for the function that would specify that any process satisfying some predicate can be chosen. Rather, this is solved by introducing nondeterminism inside of the module as in Figure 4.4. In SAL, the symbol `[]` denotes asynchronous composition. The use of `[](p: index)` provides the nondeterministic choice of one process to be executed next among those processes whose corresponding guard formula is satisfied.

```
[]
 ([] (p: index):
         c17:
           IsReader AND s.next[pID] = r4  AND
                            s.m.cnt < 0 AND s.m.set(p)
          -->  s' = (((s WITH .m.cnt := s.m.cnt + 1)
       WITH .next[pID] := r5)
    WITH .next[p] :=
                          IF s.next[p] = waitAtPm1 THEN rlseAtPm1
                          ELSE rlseAtPm2
                          ENDIF)
      WITH .m.set := remove(p, s.m.set))
[]
```

Figure 4.4: Nondeterminism inside of the `process` module

Our model of the Readers/Writers program as defined by the table in Appendix B has terminal states corresponding to the situations when all of the processes have reached the end of their code. However, some model checkers, including SAL, may

produce unsound results when checking the liveness properties of a system where not every state has at least one successor. That is why we add selfloops to those terminal states by adding a transition to the `initializator` (as in Appendix C.2), which is otherwise used for the initialization of the global variable `state`. The whole system is obtained by an asynchronous composition of `M` of `process` modules and module `initializator` as in Figure 4.5. The result of initialization is that each `process`

```
main: MODULE = initializator []
     ([] (pID : index): process[pID]);
```

Figure 4.5: The module `main`

process is instantiated with a different value of `pID`.

4.2.2 Analysis in SAL

As mentioned earlier, the current typecheck does not detect overflows. Therefore, we first have to prove that the variables of an bounded type (e.g., `semtype`) will not go over the bounds of this subrange type. This is done with the theorem `typecheck2` reproduced below:

```
typecheck2: THEOREM main
                    |- G(s.m.cnt <= 1 AND s.m.cnt >= -M+1 AND
                         s.w.cnt <= 1 AND s.w.cnt >= -M+1);
```

Here, `s.m.cnt` and `s.w.cnt` are of `semtype` type, as in Figure 4.2.1. After this check is done, we can continue the analysis with tighter bounds for the types.

The safety property from Equation 4.1 can be stated as follows:

```
safety: THEOREM main
|- G((s.wt = 0 OR s.rd = 0) AND s.wt < 2
                              AND s.rd >= 0 AND s.wt >= 0);
```

The assertion language is LTL. We decided to use a symbolic model checker, although we had a choice of infinite bounded model-checker which handles infinite state systems (unbounded types in the fields of a program state can be used, i.e., instead of the

`rdtype`, `wttype`, and `semtype` we would use integers). The infinite model-checker can provide counterexamples of a given depth or prove theorems using a generalized induction rule known as k-induction [7]. This rule first requires proving that a certain property holds in the first k steps of any execution. Then, the general step requires that, if the property is satisfied in all the executions of length k, then it will be preserved after the transition of the system to the next state. `sal-inf-bmc` was not able to prove the theorem `safety` with k-induction for k=9, which took 6667 seconds. As model-checking in our verification process would be used for refutation purposes and checking auxiliary invariants, we felt its benefits would be lost if we used `sal-inf-bmc`.

The liveness property says that all the processes will eventually complete, i.e., reach the label `EOP`. First, we check whether the transition relation is total in order to avoid unsound results. This is easily done using the SAL's `sal-deadlock-checker`. Then, we assume the weak fairness of the scheduler: if a process's enablement condition is continuously enabled, then the process will eventually execute. So, if we assume that it cannot happen that one of the non-waiting processes' enablement condition is satisfied forever, all the processes will cleanly complete. Therefore, the formalization of the liveness property under the assumption of weak fairness would be:

```
dq: THEOREM main
    |-(NOT EXISTS (i: index):
           F(G(IsReader[i] AND s.next[i] = r1 AND s.m.cnt = 1)))
           AND (NOT EXISTS (i: index):
           F(G(IsReader[i] AND s.next[i] = r1 AND s.m.cnt < 1)))
           AND ...
           => F(FORALL (k: index): IntRw(s.next[k]) = 0);
```

where the operand of the first `G` is the first "non-waiting" enablement condition from the `TRANSITION` part of `process` module, the argument of the second `G` is the second "non-waiting" enablement condition, etc.

However, the automaton for this property is too large, so that the computation runs out of memory. Therefore, we prove the liveness property as suggested in Section 2.3.2 by proving the theorem of decreasing quantity. However, for the proof

of the theorem of decreasing quantity, we found no need to define a DQ vector as suggested in Section 2.3.2 (originating from [15]), because of the assumption that no new readers/writers arrive after the initialization of the system. Moreover, if `Pos` is defined as the number of the reader/writer processes with a label other than `EOP`, then the case of a process reaching the label `EOP` ($\texttt{Pos}_1 > \texttt{Pos}_2$) can be considered as the case of decreasing one of the components of the vector IntRW defined as:

$$IntRW(next) = (IntRW(next[1]), \ldots, IntRW(next[M]))$$

Therefore, the vector $IntRW$ can be used as the decreasing quantity. We say that $IntRW$ has decreased if there is at least one element of the $IntRw$ that has decreased, while all the others have decreased or remained the same:

```
DQdecrease(s, t: state): bool = (EXISTS (i: index):
                   IntRW(t.next[i]) < IntRW(s.next[i])) AND
                   FORALL (i: index):
                   (IntRW(t.next[i]) <= IntRW(s.next[i]));
```

Note, however, that the ordering defined by `DQdecrease` is not total. We later prove that this ordering implies the `DQorder`, as originally formulated in [15] and reproduced in Section 2.3.2.

Now, the theorem of decreasing quantity as stated in Section 2.3.2, is formalized in SAL by the following three theorems:

```
dqa: THEOREM main
|- G(FORALL (u: state): (s = u AND X(s /= u))
                                    =>  X(DQdecrease(u, s)));

dqb : THEOREM main
                |-AG((FORALL (t: state): (s = t =>  EX(s /= t)))
                     OR FORALL (i: index): IntRW(s.next[i]) = 0);

dqc: THEOREM main
|- G((FORALL (i:index): IntRW(s.next[i]) = 0) =>
      FORALL (i: index): s.next[i] /= waitAtPm1 OR
```

```
      s.next[i] /= waitAtPm2 OR s.next[i] /= waitAtPwr OR
      s.next[i] /= waitAtPww);
```

Again, the automaton for the theorem `dqa` is extremely large, so that symbolic checker cannot handle it. We solve this problem by introducing the `dqmonitor` module to store the previous system state:

```
dqmonitor : MODULE =
    BEGIN
      INPUT s : state
      OUTPUT prev_state : state
      INITIALIZATION
      prev_state = ((# m := (# cnt := 1, set := {x:index | false} #),
                    w := (# cnt := 1, set := {x: index | false} #),
                    rdcnt := 0, next := [[i:index] IF i <= n THEN r1
                                                          ELSE w1
                   ENDIF],
                    rd := 0, wt := 0 #))
      TRANSITION
       prev_state' = s;
    END;
```

We then verify the appropriately modified theorem:

```
dqa_new: THEOREM main || dqmonitor
|- X(G(prev_state /= s => DQdecrease(prev_state, s)));
```

which is easily model-checked.

The theorem `dqb` is not expressible in LTL logic (because LTL cannot express the existence of a path with certain properties), so it cannot be model checked by SAL's symbolic model checker. However, this is the most general form of the theorem applicable to any concurrent system. If we bring the insight of our problem into it (meaning, state change is possible if there is at least one non-waiting process that has not reached the label `EOP` and has an enabled transition), the theorem can be model checked by checking the deadlock absence property (which we have already done) and the LTL formula:

```
dqb_new1p : THEOREM main || dqmonitor
   |- X(G((prev_state = s => (EXISTS (k: index):
              (IsReader[k] AND s.next[k] = r1 AND s.m.cnt = 1) OR
              (IsReader[k] AND s.next[k] = r1 AND s.m.cnt < 1) OR
              ....)) OR FORALL (i: index): IntRW(s.next[i]) = 0));
```

where, again, the operand of first `G` is the first "non-waiting" enablement condition from the `TRANSITION` part of `process` module, the argument of the second `G` is the second "non-waiting" enablement condition etc. The SAL model checking results are given in Table 4.2. The computation for checking `dqa` and `dqb` runs out of memory for the system consisting of 5 readers and 5 writers, and the check for safety property and `dqc` is extremely slow for the system with 6 readers and 6 writers. SAL performs worse than SPIN, due mostly to the higher complexity of SAL model and the greater size of state variable vector.

	safety		dqa_new		dqb_new1p		dqc	
	states	time(s)	states	time(s)	states	time(s)	states	time(s)
3R/2W	9961	40	34962	180	34962	190	9961	40
5R/5W	$14.9{\cdot}10^6$	2326	-	-	-	-	$14.9{\cdot}10^6$	2780
6R/6W	$0.3{\cdot}10^9$	4044	-	-	-	-	$0.3{\cdot}10^9$	4044
7R/7W	$6.1{\cdot}10^9$	55627	-	-	-	-	$6.1{\cdot}10^9$	55627
15R/10W	-	-	-	-	-	-	-	-

Table 4.2: SAL model checking results

4.2.3 Summary

In summary, we were able to model-check our model for safety and clean completion (using the theorem of decreasing quantity). For the theorem of decreasing quantity, we had to modify the second part of the theorem, since it initially was not expressible in LTL. Moreover, since the current version of SAL is missing a full typechecker, we were not able to check our specification for coverage and consistency, and had to perform some additional checks (e.g., that the variables of a certain subrange type will not cross the bounds of that type).

While SAL's performance on the more detailed model of the problem lags behind the performance of SPIN, we note that the SAL model as described here will be used in the next chapter for model-checking all the auxiliary invariants discovered by deduction in PVS.

Chapter 5

Theorem Proving in PVS

In the previous chapter, we formalized the Readers/Writers problem with a fixed number of readers/writers, rewritten as in [15] using the SAL specification language. Safety and liveness properties were automatically proven using the SAL symbolic model checker. In this chapter, we first try to verify the hand-written proof of the full system with an arbitrarily large number of readers/writers from [15] and then give a significantly more automated proof of the same problem combining theorem proving in PVS and model checking in SAL.

5.1 The Theory Hierarchy

The theory hierarchy diagram is given in Figure 5.1, where $A \longrightarrow B$ denotes "Theory `A` is imported by theory `B`". The `decl` theory contains the type definitions, functions, etc. The theory `conds` imports the `decl` and defines the headers of the table given in the theory `transition`. The `getinv` theory contains mostly unprovable theorems, used for reaching the inductive invariant. The `invj`, `invj1`, and `cardsem` theories define the invariants and theorems needed to prove the safety property. The `dq`, `dqb`, `dqbfinal`, and `ordering` contain the definitions of the invariants and theorems needed to prove the clean completion property.

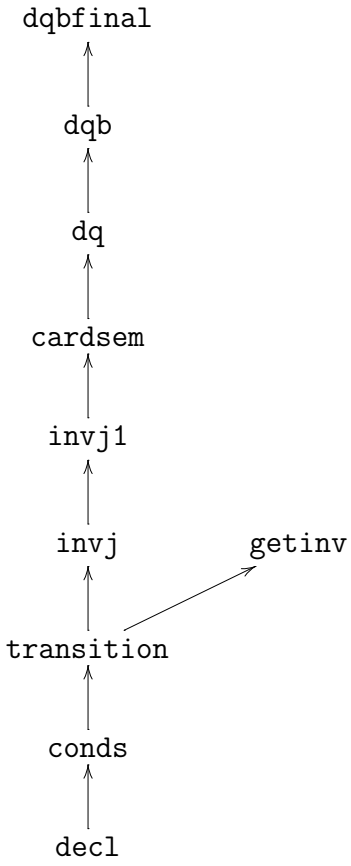

Figure 5.1: The theory hierarchy

5.2 The `decl` Theory

The `decl` theory in Figure 5.2 contains the definitions of types, functions, etc. The program state is defined as the record type `state`. However, we also needed the predicate subtype `stateneop`, which we use to help reflect the fact that a process that has terminated (reached the label `EOP`) cannot become the executing process.

The process chosen in the execution of the program is identified by an index variable `pID` (the variables are taken from [15]). A global variable of the type `state` contains the resources shared by all the processes: semaphores `m` and `w`, then counters `rd`, `wt`, `rdcnt`, array of processes' labels `next` and `pID`, the identifier of the currently executing process. Indices of the array are the process identifiers. The predicate

```
M: posnat
ntype: TYPE = {i: nat | i <= M}
index: TYPE = {i: ntype | i >= 1} CONTAINING 1
n: ntype
label: TYPE = {r1, waitAtPm1, rlseAtPm1, r2, r3, waitAtPwr,
               rlseAtPwr, r4, r5, r6, waitAtPm2, rlseAtPm2,
               r7, r8, r9, w1, w2, w3,
               waitAtPww, rlseAtPww, EOP}
x: VAR label
rlabel?(x): bool = (x = r1 or x =  waitAtPm1 or
                           x = rlseAtPm1 or  x = r2 or
                           x = r3 or x = waitAtPwr or
                           x = rlseAtPwr or x = r4 or
                           x = r5 or x = r6 or
                           x = waitAtPm2 or x = rlseAtPm2 or
                           x = r7 or x = r8 or
                           x = r9 or x = EOP)
wlabel?(x): bool = (x = w1 or x = w2 or x = w3 or
                           x =  waitAtPww or x = rlseAtPww or
                           x = EOP)
IsReader(i: index): bool = (i <= n)
ar: TYPE = {a: [index -> label] | forall (i: index):
                    ((IsReader(i) =>  rlabel?(a(i))) and
                     (not IsReader(i) => wlabel?(a(i))))}
importing finite_sets[index]
sem: TYPE = [#cnt: integer, set: finite_set#]
state: TYPE = [#
               pID: index,
               m: sem,
               w: sem,
               rdcnt: int,
               next: ar,
               rd: int,
               wt: int  #]
stateneop: TYPE = {s: state | next(s)(pID(s)) /= EOP}
```

Figure 5.2: Theory `decl`

`IsReader` takes as an argument a variable of type `index` and is true if the process in question is a reader process ($i \leq n$), and false if the process is a writer process ($n < i \leq M$), where ($0 \leq n \leq M$).

This theory also contains a definition of the function `IntRW` (also taken from [15]), used for proving the clean completion of the program. It maps all the possible values of the variable next to integers as in Figure 5.3.

```
IntRW(x: label): int =
 COND
        x=r1    ->     15,
        x=waitAtPm1 -> 14,
        x=rlseAtPm1 -> 13,
        x=r2   ->      12,
        x=r3   ->      11,
        x=waitAtPwr -> 10,
        x=rlseAtPwr -> 9,
        x=r4 ->        8,
        x=r5  ->       7,
        x=r6 ->        6,
        x=waitAtPm2 -> 5,
        x=rlseAtPm2 -> 4,
        x=r7   ->      3,
        x=r8    ->     2,
        x=r9   ->      1,
        x=w1   ->      5,
        x=waitAtPww -> 4,
        x=rlseAtPww -> 3,
        x=w2   ->      2,
        x=w3   ->      1,
        x=EOP  ->      0

 ENDCOND
```

Figure 5.3: PVS definition of the function `IntRW`

5.3 The `table` Theory

The tabular representation of the Readers/Writers rewritten program in Appendix B (originally taken from [15]) is represented as a theory in PVS. Part of this theory is shown in Figure 5.4.

```
trans(s : {s:stateneop |
       NOT (p1(s) or p7(s) or p10(s) or p12(s)
       or p15(s) or p19(s) or p25(s) or p28(s) or p30(s)
       or p33(s) or p39(s))}, t: state): bool =
 LET k: index = pID(s) IN
 table
 %-------------------------------------------------------------||
 |p1(s)|                                                       ||
 %-------------------------------------------------------------||
 |p2(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) - 1 and        %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall (j:index): (j= k and next(t)(j) = r2) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
 %-------------------------------------------------------------||
 .
 .
 .
```

Figure 5.4: Tabular representation of Readers/Writers problem in PVS

The table from [15] is modeled with a transition relation `trans`. The relation `trans(s, t)` evaluates to `TRUE` if one of the guard conditions `p1(s)` to `p41(s)` (whose definitions are given in Appendix C.3) holds and the program can make the transition from state `s` to `t`. Note that the PVS table is the original table transposed for readability in the PVS ASCII text input format.

The first argument of the `trans` is of the type `stateneop`. In order to make the relation total, the first argument is subtyped to reflect the fact that some states satisfying certain predicates (`p1(s)`, `p7(s)`, `p10(s)`, `p12(s)`, `p15(s)`, `p19(s)`, `p25(s)`,

`p30(s)`, `p33(s)`, `p39(s)`) can never be reached. The table entries corresponding to those predicates are left blank. PVS generates TCCs that requires a user to prove that the states satisfying those predicates are indeed unreachable (see Subsection 5.5.1).

The disjointness obligation for the table `trans` is automatically discharged by PVS, and the completeness obligation is discharged after making the type constraints of `next` explicit.

5.4 Verifying the Hand-Written Proof

The requirements of the Readers/Writers program say that only one writer can be active while no reader is reading or one or more readers can read concurrently while no writer is writing. This can be stated as in [15]:

$$(rd = 0 \; or \; wt = 0) \; and \; wt < 2$$

This global invariant is defined in PVS as two invariants `rp1` and `rp2`:

```
t: VAR state
rp1(t): bool = wt(t) = 0 or rd(t) = 0
rp2(t): bool = wt(t) < 2
```

The initial condition for the system is given by:

```
initcond(t): bool = cnt(m(t)) = 1 and empty?(set(m(t))) and
                    cnt(w(t)) = 1 and empty?(set(w(t))) and
                    rd(t) = 0 and wt(t) = 0 and rdcnt(t) = 0 and
                    (forall (i: index): (i <= n  and  next(t)(i) = r1)
                    or (i > n  and  next(t)(i) = w1))
```

In initial state, the semaphore `m` is available (`cnt(m(t)) = 1`), and there are no processes waiting for it (`empty?(set(m(t)))`). The same holds for the semaphore `w`. The initial values of `rd`, `wt`, and `rdcnt` are zero, and the reader and writer processes are at the `r1` and `w1` label, respectively.

Strictly following the manual proof of [15], we first try to prove `rp1`, by proving that it is true after initialization:

```
initrp1: theorem  initcond(t) => rp1(t)
```

and, row by row (or, column by column, for the original table), that it is preserved after every statement in the program.

One of the theorems from the manual proof of [15] we are to prove is:

```
s: VAR stateneop
t: VAR state
cc14rp1: theorem p14(s) and rp1(s) and trans(s, t) implies rp1(t)
```

The previous theorem states that if the `p14` guard condition is satisfied, `rp1` predicate holds, and the system makes a valid transition, the `rp1` should also hold in the new state.

Starting the PVS theorem prover gives three unprovable sequents, one of which is the following:

```
{-1}  (pID(s!1) <= n)
{-2}  rlseAtPwr?(next(s!1)(pID(s!1)))
{-3}  rd(s!1) = 0
{-4}  rdcnt(t!1) = rdcnt(s!1)
{-5}  rd(t!1) = 1
{-6}  wt(t!1) = wt(s!1)
{-7}  cnt(m(t!1)) = cnt(m(s!1))
{-8}  set(m(t!1)) = set(m(s!1))
{-9}  cnt(w(t!1)) = cnt(w(s!1))
{-10} set(w(t!1)) = set(w(s!1))
{-11} pID(t!1) = pID(s!1)
{-12} r4?(next(t!1)(pID(s!1)))
  |-------
{1}   wt(s!1) = 0
```

By analyzing the sequent shown above, we realize that it is requiring us to show that if a process can get a permission to enter a critical section (`rlseAtPwr?(next(s!1)(pID(s!1)))`) then it must be that the critical section is empty; therefore, there are no writers already writing (`wt(s!1) = 0`). The same thinking can be applied to the remaining two sequents.

Therefore, the proposed new, auxiliary invariant would be:

```
inv14(t): bool = forall (i: index): next(t)(i) = rlseAtPwr
                                     => wt(t) = 0
```

It states that if a reader process has acquired permission to enter critical section, it must be the case that there are no writers in it; otherwise, it would happen that both readers are reading and a writer is writing in the critical section at the same time.

The manual proof from [15] used two different invariants, denoted `V8` and `V10` (whose definitions can be found in the list of invariants from the manual proof in Appendix C.5):

```
c14rp1: theorem  p14(s) and rp1(s) and V8(s) and V10(s) and
                      trans(s, t) implies rp1(t)
```

This theorem, however, could not be proven in PVS. The unprovable sequent indicated the `inv14` invariant again. By investigating the manual proof, we came to the conclusion that the error was made because one branch of the proof was not explored at all: the first disjunct of the consequent of formula `V8` was left out during the course of the proof. This corresponds to the case when there is a writer writing, and a reader got permission to enter the same critical section. Obviously, this is not possible, but this conclusion does not follow from the facts provided in the theorem `c14rp1`.

If we continue proving `rp1` for the remaining rows, discovering more invariants, then those discovered invariants should be proven themselves. However, proving the auxiliary invariants of the form $(\exists i : (i = pID(t) \wedge next(t)(i) = l)) \Rightarrow P(t)$, where P is a predicate on the global state of the system t, and l is some label, discovered a more serious flaw of the proof: only part of the transition relation was explored. Model checking in SAL confirmed this conclusion. The manual proof actually considered the relation from the table with an additional assumption: the `pID` of the currently executing process does not change after the transition of the program to the next state. (Even for this modified relation we found two invariants in [15] needed strengthening (`V10, V15`)). Since only a part of the relation was explored, some of the invariants found by hand do not hold in all the states of the system with the full transition relation. For instance, the invariant `V12` from [15]:

```
V12(t): bool = (exists (i: index): i = pID(t) and
```

```
                (next(t)(i) = r1 or next(t)(i) = rlseAtPm1 or
                 next(t)(i) = r2 or next(t)(i) = r4 or
                 next(t)(i) = r5 or next(t)(i) = r6 or
                 next(t)(i) = rlseAtPm2 or next(t)(i) = r7 or
                 next(t)(i) = r9)) implies rd(t) = rdcnt(t)
```

claims that 'if any reader when executed has a label of e.g. `r1`, then it must be that `rd` is equal to `rdcnt`'. But, this is not the case. In fact, it can happen that there is another process whose label is e.g. `r3`, so that at a state of executing the process with `r1` label, `rdcnt` would be greater than `rd`. The counterexample for the system with two readers and two writers was generated by model checking a modified version of the invariant in SAL. The invariant is modified, because we did not need to explicitly model the `pID` of the currently executing process in SAL, since the model checker explores all the possible subsequent states of a state, corresponding to different processes being chosen to be executed next, the validity of the counterexample given below is preserved. The modified invariant is:

```
V12(t): bool = (exists (i: index):
                (next(t)(i) = r1 or next(t)(i) = rlseAtPm1 or
                 next(t)(i) = r2 or next(t)(i) = r4 or
                 next(t)(i) = r5 or next(t)(i) = r6 or
                 next(t)(i) = rlseAtPm2 or next(t)(i) = r7 or
                 next(t)(i) = r9)) implies rd(t) = rdcnt(t)
```

Counterexample generated by SAL is given below:

$$(\mathrm{r1}, \mathrm{r1}, \mathrm{w1}, \mathrm{w1}, 0, 0) \overset{pID=1}{\rightarrow} (\mathrm{r2}, \mathrm{r1}, \mathrm{w1}, \mathrm{w1}, 0, 0) \overset{pID=1}{\rightarrow} (\mathrm{r3}, \mathrm{r1}, \mathrm{w1}, \mathrm{w1}, 0, 1)$$

The 6-tuples represent the relevant part of the program state: $(next[1], next[2], next[3], next[4], rd, rdcnt)$.

To gain a better understanding of what the PVS version of the manual proof really proved, take a look at the `V9` invariant, also from [15]:

```
V9(t): bool = (exists (i: index): i = pID(t) and
                next(t)(i) = r7) => rdcnt(t) > 0
```

which should actually be

```
V9_new(t): bool = (exists (i: index):
                next(t)(i) = r7) => rdcnt(t) > 0
```

The PVS version of the manual proof proved that the predicate `V9` is invariant if there is exactly one process with label `r7` (a process having a label `r7` is in the critical section of semaphore `m`) in state `s` and that is the process currently executing, or there are no processes at the `r7` label in state `s`. It has not, however, discharged the proof obligations in the case where e.g., there is at least one process with the `r7` label in state `s`, but any other process is chosen to be executed. In this case, there cannot exist a process whose execution would decrease `rdcnt`. If this was the case, it would mean that there exists another process with label `r7`, which is a contradiction, because there cannot be two processes in the critical section of semaphore `m`. Therefore, we need another invariant:

```
  CS1(t): bool = (forall (i, j: index): CS1pred(t, i)
                                    and CS1pred(t, j) => i = j)
```

where

```
  i: VAR index
  CS1pred(t, i): bool =  next(t)(i) = rlseAtPm1 or
               next(t)(i) = r2 or next(t)(i) = r3 or
               next(t)(i) = r4 or next(t)(i) = rlseAtPm2 or
               next(t)(i) = r7 or next(t)(i) = r8 or
               next(t)(i) = r9 or next(t)(i) = waitAtPwr or
               next(t)(i) = rlseAtPwr
```

It says that it cannot be the case that there is more than one process in the critical section of semaphore `m`. The same thing, of course, holds for semaphore `w`. This will be discovered by PVS, as suggested in the next section.

5.5 Verification in PVS Revisited

In this subsection we give a significantly automated proof for both safety and liveness properties. While the PVS proof still mimics the manual proof's "divide and conquer"

technique by considering the proof in a row by row case, the process is significantly automated. Rather than having to explicitly state and prove a theorem for each row of the table, proof tactics have been developed that examine the structure of the table and decompose the complete proof obligation into proof subgoals, one for each row of the table.

5.5.1 Proof of the Safety Property

First, we change the requirement from [15] as indicated in Subsection 4.1.2:

```
rp(t): bool = (wt(t) = 0 or rd(t) = 0) and
               wt(t) < 2 and rd(t) >= 0 and wt(t) >= 0
```

Secondly, we prove the global property for the whole table at once, rather than using "a theorem per row" approach:

```
crp11: lemma  forall t: (initcond(t) => rp(t))
     and  forall s, t: ((rp(s) and trans(s, t)) => rp(t))
```

Attempt to prove the `crp11` theorem with `(GRIND)` after making the type constraints of `next` explicit and instantiating the corresponding formula with `pID(s!1)` yields 210 subgoals (it takes less than 5 minutes), one of which is shown here:

```
crp11.2.1 :
{-1}  pID(t!1) <= M
{-2}  pID(t!1) >= 1
{-3}  r1?(next(s!1)(pID(s!1)))
{-4}  wt(s!1) = 0
{-5}  rd(s!1) >= 0
{-6}  rdcnt(t!1) = rdcnt(s!1)
{-7}  rd(t!1) = rd(s!1)
{-8}  set(m(t!1)) = set(m(s!1))
{-9}  cnt(m(t!1)) = cnt(m(s!1))
{-10} cnt(w(t!1)) = 1 + cnt(w(s!1))
{-11} set(w(s!1))(pID(t!1))
{-12} set(w(t!1)) = remove(pID(t!1), set(w(s!1)))
{-13} wt(t!1) = -1
{-14} (p!1 = pID(t!1))
```

```
{-15} waitAtPww?(next(s!1)(pID(t!1)))
{-16} rlseAtPww?(next(t!1)(pID(t!1)))
  |-------
{1}   pID(s!1) > n
{2}   cnt(m(s!1)) = 1
{3}   EOP?(next(t!1)(pID(t!1)))
{4}   cnt(m(s!1)) < 1
{5}   (pID(t!1) = pID(s!1))
```

The lines {2} and {3} of the previous sequent combined require that `cnt(m(s!1))` cannot be greater than 1. This should always hold according to the specification of the semaphore. So, we need to strengthen our property with `S1(s)`: `S1(s)`:

```
S1(t): bool = cnt(m(t)) <= 1
```

Most of the subgoals are repeated, so it is not as hard to analyze the sequents as it may appear at first. The number of unprovable goals drastically decreases in the next iterations.

After considering all of the 210 subgoals, we obtained a set of twelve invariants given in Appendix C.3 to be used to strengthen the initial invariant, so we now prove the stronger property:

```
 s: VAR stateneop
 t: VAR state
 ind1(t): bool = rp(t) and S1(t) and S2(t) and S31(t)
                 and S32(t) and S41(t) and S5(t)
                 and S6(t) and S7(t) and S81(t)
                 and S82(t) and S91(t) and S101(t)
 crpind1: lemma (forall t: initcond(t) => ind1(t))
                            and forall s, t: (ind1(s)
                            and trans(s, t) => ind1(t))
```

Using the knowledge gained from the analysis in the previous section, we designed a strategy to prove this lemma, or, rather, gain new invariants. Branches of the proof corresponding to the invariants that are universally quantified on `i` are split into two cases. First case, for `i!1 = pID(s!1)`, we apply `GRIND`, and contemplate

the invariants from the unprovable sequents. However, we choose to skip the case for `i!1 /= pID(s!1)`, since the vast majority of the failed goals corresponding to this branch can be subsumed into an invariant saying that there cannot be more than one process in the critical section of semaphore `m` and semaphore `w`. One of the sequents gained from these branches (`i!1 /= pID(s!1)`) is the following:

```
crpind1.2.4.2.2 :

{-1}  r4?(next(s!1)(i!1))
{-2}  r4?(next(s!1)(pID(s!1)))
{-3}  wt(s!1) = 0
{-4}  rd(s!1) >= 0
{-5}  (cnt(w(s!1)) <= 1)
{-6}  (rdcnt(s!1) >= 0)
{-7}  cnt(m(s!1)) = 0
{-8}  rdcnt(t!1) = rdcnt(s!1)
{-9}  rd(t!1) = rd(s!1)
{-10} wt(t!1) = 0
{-11} cnt(m(t!1)) = 1
{-12} set(m(t!1)) = set(m(s!1))
{-13} cnt(w(t!1)) = cnt(w(s!1))
{-14} set(w(t!1)) = set(w(s!1))
{-15} next(t!1)(i!1) = next(s!1)(i!1)
  |-------
[1]   i!1 = pID(s!1)
{2}   r1?(next(s!1)(pID(s!1)))
{3}   waitAtPm1?(next(s!1)(pID(s!1)))
{4}   rlseAtPm1?(next(s!1)(pID(s!1)))
{5}   r2?(next(s!1)(pID(s!1)))
{6}   r3?(next(s!1)(pID(s!1)))
{7}   waitAtPwr?(next(s!1)(pID(s!1)))
{8}   rlseAtPwr?(next(s!1)(pID(s!1)))
{9}   pID(s!1) > n
{10}  cnt(w(s!1)) = 1
```

The invariant corresponding to this sequent says that there cannot be two different processes at one time with the label `r4` (a process whose label is equal to `r4` is in the critical section of semaphore `m`). This invariant is a part of one of the two 'semaphore' invariants `CS1` and `CS2`, whose definitions are given in Appendix C.4.

We continue on with strengthening the property using the same tactic without using semaphore invariants, until we prove that the conjunction of the global property and the newly found invariants is inductive for the branches corresponding to `i!1 = pID(s!1)`. We needed six iterations to reach inductivity. Every iteration contains the following steps:

1. We formalize the theorem in PVS that states that a property includes all the initial states and is closed under all possible transitions.

2. If the proof fails, we obtain the new potential auxiliary invariants indicated by unprovable sequents.

3. New invariants are model checked in SAL.

4. The desired property now becomes the conjunction of the old property and newly found ones. However, we choose to prove only the properties that were not proved (for `i!1=pID(s!1)`) in the previous iteration and the newly found ones.

As indicated in step 3, all the auxiliary invariants are first model checked. The list of those can be found in Appendix C.4. The verification using model checking being fully automatic made the checking of the auxiliary invariants fast and easy. It increased the confidence in our PVS deductive analysis and provided fast discovery of "fake" invariants (proposed invariants originating in a mistake made while contemplating the invariant from the characteristic equation of an unprovable sequent). The mistake would, obviously, be caught by PVS, but at best in the next iteration (which is still time-consuming and not as obvious), and under the assumption that the SAL and PVS models are equivalent.

Now, we are to prove that all those auxiliary invariants are invariants. We came up with another four auxiliary invariants, corresponding to the cases where the label of a process is changed by executing another process (a process is releasing semaphore, and the other process can enter the critical section). We ended up with 42 invariants all together. Proofs of the 'semaphore' invariants are divided into lemmas because of the time and memory constraints. Special proof tactics were also written for those lemmas.

All the strategies are in Appendix C.3. They all use a "divide and conquer" policy: every proof is split into 31 branches (where 31 is the number of non-blank table columns). We did not use PVS' built-in strategy `bddsimp` (propositional simplification) to break down proof goals; in the general case, the use of `bddsimp` would result in many more goals than the number of rows - those would correspond to the disjuncts in the grid cells of the table. Obtained goals are then tackled with the same tactic. This tactic is chosen so that the degree of the automaton of the process, and memory and time consumption, are balanced. The vast majority of the invariant proofs (around 80%) are completely automated using those strategies; for the rest, after applying a corresponding strategy, the unprovable sequents of some branches clearly indicate the further steps, so that a minimal level of human insight is needed to help finish up the proofs. The achieved run-times of the proofs can be decreased with more human interaction. The higher level of human guidance would involve choosing the invariants needed for a particular auxiliary invariant proof (since not all the invariants in the inductive invariant are needed to prove each auxiliary invariant) and would substantially decrease the times.

At the end, we are to prove the proof obligations for each of the final lemmas, e.g. for the `S121` invariant:

```
% Subtype TCC generated (at line 266, column 38) for  s
    % expected type  {s: state1 |
      %     NOT (    p1(s) OR p7(s) OR p10(s) OR p12(s)
      %         c\ OR p15(s) OR p19(s) OR p25(s) OR p28(s)
      %          OR p30(s) OR p33(s) OR p39(s))}
  % untried
crpind121_TCC1: OBLIGATION
  (FORALL t: initcond(t) => S121(t)) IMPLIES
   (FORALL (s, t1):
      indc(s) IMPLIES
       NOT (    p1(s) OR p7(s) OR p10(s)
             OR p12(s) OR p15(s) OR p19(s)
             OR p25(s) OR p28(s) OR p30(s) OR p33(s) OR p39(s)));
```

for which strategies are also written. These obligations require us to prove that the system, so far described with the invariant `indc`, can never reach a state which satisfies any of the `p1` to `p39` predicates.

The process of proving the safety property as proposed is largely an automated one. First, the unprovable sequents as the indicators of the invariants needed are obtained automatically, using specially written strategies. However, human insight is needed to determine the invariants from these unprovable sequents. The process of proving that those new invariants are invariants indeed is completely automated for the majority of invariants and takes 10 minutes on average (except for the "semaphore" invariants which take few hours). The semaphore invariants are system specific, but could, in the future be generalized in a "semaphore" theory.

5.5.2 Proof of the Theorem of Decreasing Quantity

We use the vector `IntRW` as a decreasing quantity as explained in Subsection 4.2.2. We redefine the predicate `DQdecrease` in PVS as:

```
s:VAR stateneop
t: VAR state
DQdecrease(s, t): bool = (exists i: IntRW(next(s)(i)) >
                          IntRW(next(t)(i))) and
                          (forall i: IntRW(next(s)(i)) >=
                          IntRW(next(t)(i)))
```

The theorem of decreasing quantity is given in Section 2.3.2 (originally taken from [15]).

We first formalize the first part of the theorem of decreasing quantity. We prove that every two states, `s` and its next state `t`, that differ in at least one field other than the `pID` field, satisfy `DQdecrease(s, t)`:

```
s:VAR stateneop
t: VAR state
dqa: theorem indc4(s) => (trans(s, t) and  not (m(s) = m(t) and
               w(s) = w(t) and rdcnt(s) = rdcnt(t) and
```

```
(forall i: next(s)(i) = next(t)(i)) and
rd(s) = rd(t) and wt(s) = wt(t)) => DQdecrease(s, t))
```

The predicate `indc4` (defined in Appendix C.3 in the PVS file `cardsem`) is the inductive invariant found in the safety property proof. Therefore, it contains all the information on our state space that we have obtained so far.

Part `b)` of the theorem of decreasing quantity states that it is either the case that the decreasing quantity has reached zero, or that there is a possible state change (other than change of `pID`). We formalize it as:

```
s1, t, u: VAR state
dqb: lemma forall s1: (indc4(s1) =>
                       ((forall i: IntRW(next(s1)(i)) = 0) or
                       (exists t: (trans(s1, t) and
                       (not (m(s1) = m(t) and
                       w(s1) = w(t) and rdcnt(s1) = rdcnt(t) and
                       (forall i: next(s1)(i) = next(t)(i)) and
                       rd(s1) = rd(t) and wt(s1) = wt(t)) or
                       (exists u: (trans(t, u) and not (m(t) = m(u) and
                       w(t) = w(u) and rdcnt(t) = rdcnt(u) and
                       (forall i: next(t)(i) = next(u)(i)) and
                       rd(t) = rd(u) and wt(t) = wt(u))))))))
```

The `dqc` part of the decreasing quantity theorem says that if the decreasing quantity has reached zero, then there are no waiting processes:

```
t: VAR state
dqc: theorem indc4(t) => (forall (i: index): IntRW(next(t)(i)) = 0)
     implies (forall (i: index): (next(t)(i) /= waitAtPm1
     and  next(t)(i) /= waitAtPm2 and next(t)(i) /= waitAtPwr
     and  next(t)(i) /= waitAtPww))
```

Proving the `dqb` theorem required the additional strengthening of the invariant that was found sufficient for proving the safety property. Reduction of the state space from `indc4` to `indc8` (definition of `indc8` is given in Appendix C.3 in the PVS

file `dqbfinal`) would have required many iterations, if we were to use exclusively the failed goals in PVS in order to come up with the invariants. These iterations were skipped by human intervention with significant help of the SAL model checker. We needed 12 new invariants. The proofs for those invariants are not completely automated, since the proofs are distinct, so we did not feel that we would benefit from writing strategies. On the other hand, the theorems `dqa` and `dqc` were easily proven. Finally, we proved that the partial order `DQdecrease` implies the total order `DQorder` from the original theorem of decreasing quantity from [15].

5.6 Summary

We formalized the Readers/Writers problem rewritten into a table as in [15] in PVS. The verification of the manual proof of the safety property from [15] using a combination of theorem proving in PVS and model checking in SAL has discovered mistakes in the manual proof. This was a rather useful guide to some of the problems one might encounter in inspecting a concurrent problem using the method proposed in [15], and provided an understanding of the importance of automation in the process. Finally, a significantly automated proof of the safety property was given using PVS proof tactics, while the proof of the clean completion property required significant human assistance.

Chapter 6

Conclusion

6.1 Summary

The state explosion problem limits the scope of use of model checking. For large state spaces theorem proving still remains the inevitable option. Many techniques have tried to combine the automaton of model checking and generality of theorem proving. The central role of our approach is given to theorem proving. Model-checking is used for refutation purposes: as a debugging tool for the original program (SPIN), or the program rewritten into a table (SAL) in case SPIN missed on finding some bugs, or they were introduced while rewriting the program into a table. Moreover, SAL proved to be extremely useful for checking the auxiliary invariants.

We believe that the contributions of our work are the following:

- We provided partial automaton of the inspection process of [15].

- We provided the basis for automated reasoning about concurrent programs based on tabular expressions. We believe that many of the issues dealt in the analysis of the Readers/Writers example in this thesis will reappear in the verification of other concurrent problems using the same inspection approach. E.g., the use of 'pregenerated' invariants inherent to the synchronization (communication) mechanisms used would significantly reduce the time needed to obtain the final, inductive invariant. Moreover, as the `next` variable is inherent to this inspection process, the reappearance of the universally quantified implications

of the form $\forall i : (next(t)(i) = l \Rightarrow P(t))$, where P is a predicate on the global state of the system t, and l is some label, is predictable. Therefore, the tactics written for some types of invariant are reusable to a certain extent.

- We illustrated the necessity of the computer-aided verification of the concurrent systems in inspection of [15] by automating the manual proof of the safety property of the Readers/Writers problem (as in [15]). The proposed combination of theorem proving and model checking discovered several inadvertent and one systematic mistake in the manual proof. More precisely, model checking itself indicated that some of the invariants found in manual proof were not the invariants of the program. Theorem proving offered a better insight into the depth of the systematic mistake made: it showed exactly what part of the transition relation was left out by the mistake.

- A detailed example of the computer-aided verification of the concurrent programs with arbitrarily large number of processes is given.

- Theorem proving and model checking were successfully combined. Two model checking tools (one of which is specialized for models of concurrency, the other one with an input language very close to that of the theorem prover) were used for model checking the classical concurrent program. Ideally, we would want to have used only one model checking tool, which would be specialized for concurrency and offer a successful combination with a theorem prover (e.g., capability to export from one to another).

- Our approach pointed out the need for a symbolic analysis framework that would successfully integrate model checking, theorem proving, invariant generation and abstraction.

Although it provided for a fast and automatic finding of bugs, model checking was not sufficient to prove the correctness of the systems with arbitrary number of processes: only the instances of the system could be checked. This is why theorem proving was needed.

We used a PVS construct for tabular specification in order to specify our program. The construct generated a proof obligations to ensure that the row conditions are

disjoint and complete. Since the construct is highly integrated with other capabilities of PVS, we were able to prove the invariant property and the theorem of decreasing quantity. Failed proofs indicated additional invariants needed to prove the invariant of the program. Formalizing the same problem in SAL using the symbolic model-checker provided checking the auxiliary invariants using the symbolic model-checker (but, for the system with the fixed number of processes) and increased the confidence in our deductive analysis.

6.2 Limitations and Future Work

In our verification of Readers/Writers program we used a specific implementation of semaphore, as specified in Appendix A. For future work, we would suggest investigating the possibility of using the specification of a synchronization primitive rather than its implementation. This should enable us to use the same proof for different implementations of a synchronization primitive, while only verifying its specification axioms as given in [10], against a particular implementation.

The process of finding an invariant strong enough is crucial in order to prove safety property and theorem of decreasing quantity, as already concluded in [15]. Finding the auxiliary invariants and proving that those are indeed the invariants of the system was automated as far as possible using special tactics based on PVS' built-in decision procedures. The proof of the majority of invariant lemmas is completely automated and took as much as 10 minutes on average. Substantial human guidance can be used to decrease the times. We believe that the planned integration of PVS and ICS decision procedures [9] will significantly reduce the time needed to complete the proof.

Obviously, the translator from SAL to PVS would make the process more effective. The further development of SAL as a powerful tool combining the theorem proving, model checking, abstraction and invariant generation will offer the means of the enhanced analysis, including the automated invariant generation and the existential abstraction as suggested in [34]. We believe that the lessons learned during the course of this thesis will offer a valuable guidance on combining tables and automated verification for the successful inspection of concurrent systems.

Bibliography

[1] D. Berry, "Formal specification and verification of concurrent programs," Tech. Rep. SEI-CM-27-1.0, Sortware Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[2] E. M. Clarke, O. G. Jr., and D. A. Peled, Model Checking. Cambridge, Massachusetts: The MIT Press, 2001.

[3] B. Cohen, W. T. Harwood, and M. I. Jackson, The Specification of Complex Systems. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[4] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with readers and writers," Commun. ACM, vol. 14, no. 10, pp. 667–668, 1971.

[5] L. de Moura, "SAL: Tutorial," tech. rep., Computer Science Laboratory, SRI International, Menlo Park, CA, August 2004.

[6] L. de Moura, S.Owre, and N. Shankar, "The SAL language manual," Tech. Rep. CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, 2003.

[7] L. M. de Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification (extended abstract, category A).," in CAV, pp. 14–26, 2003.

[8] E. W. Dijkstra, A Discipline of Programming. Englewood Cliffs, NJ: Prentice Hall, 1976.

[9] Formal Methods Program, "Formal methods roadmap: PVS, ICS, and SAL," Tech. Rep. SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 2003.

[10] A. N. Habermann, "Synchronization of communicating processes," Commun. ACM, vol. 15, no. 3, pp. 171–176, 1972.

[11] K. Havelund and N. Shankar, "Experiments in theorem proving and model checking for protocol verification," in FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods, (London, UK), pp. 662–681, Springer-Verlag, 1996.

[12] C. A. R. Hoare, "Communicating sequential processes," Comm. ACM, vol. 21, no. 8, pp. 667–677, 1978.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," Comm. ACM, vol. 12, no. 10, pp. 576–580, 583, October, 1969.

[14] G. J. Holzman, The SPIN Model Checker. Addison-Wesley, 2003.

[15] X. J. Jin, "Use of tabular expression in the inspection of concurrent programs," Master's thesis, McMaster University, December 2004.

[16] L. Lamport, "Win and sin: Predicate transformers for concurrency," Tech. Rep. 17, Digital Systems Research Center, Palo Alto, CA, May 1987.

[17] M. Lawford, P. Froebel, and G. Moum, "Application of tabular methods to the specification and verification of a nuclear reactor shutdown system." To appear in Formal Methods in System Design, 2004.

[18] M. Lawford, J. McDougall, P. Froebel, and G. Moum, "Practical application of functional and relational methods for the specification and verification of safety critical software," in Proceedings Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000 (T. Rus, ed.), vol. 1816 of LNCS, pp. 73–88, Springer, 2000.

[19] M. Lawford and W. Wonham, "Equivalence preserving transformations of timed transition models," IEEE Trans. Autom. Control, vol. 40, pp. 1167–1179, July 1995.

[20] R. Milner, A Calculus of Communicating Systems, vol. 92 of Lecture Notes in Computer Science. Springer, 1980.

[21] R. Milner, Communication and Concurrency. Prentice–Hall, 1989.

[22] L. Moura, S. Owre, and N. Shankar, "The SAL language manual," tech. rep., SRI, Menlo Park, California, August 2003.

[23] T. Murata, "Petri nets: Properties, analysis and applications," Proceedings of the IEEE, vol. 77, no. 4, pp. 541–580, April, 1989.

[24] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," Comm. ACM, vol. 19, no. 5, pp. 279–285, May, 1976.

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

[26] D. L. Parnas, "Inspection of safety-critical software using program-function tables," in IFIP Congress(3), pp. 270–277, 1994.

[27] D. L. Parnas, J. Madey, and M. Iglewski, "Precise documentation of well-structured programs," IEEE Trans. Softw. Eng., vol. 20, no. 12, pp. 948–976, 1994.

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

[29] D. Parnas, G. Asmis, and J. Madey, "Assessment of safety-critical software in nuclear power plants," Nuclear Safety, vol. 32, no. 2, pp. 189–198, April–June 1991.

[30] J. Rushby, "Tutorial introduction to mechanized formal analysis using theorem proving, model checking and abstraction," tech. rep., SRI, Menio Park, California, May 2003.

[31] S. Rusovan, "Inspecting the source code that implements the PPP protocol in linux," Tech. Rep. 19, SQRL, McMaster University, Jan. 2004.

[32] A. J. V. Schouwen, "The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems," Tech. Rep. 242, Communications Research Laboratory, McMaster University, May 1992.

[33] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, "PVS prover guide - version 2.2."

[34] N. Shankar, "Combining theorem proving and model checking through symbolic analysis," in CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory, (London, UK), pp. 1–16, Springer-Verlag, 2000.

[35] S.Owre, J. Rushby, and N. Shankar, "Analyzing tabular and state-transition requirements specifications in PVS," Tech. Rep. CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995.

Appendix A

Specification of P/V Semaphore Operations

The following tabular specification of P/V operations of a semaphore is taken from [15].

Figure A.1 represent the tabular representation of $P(sem)$ operation of sem semaphore. x represents the label of currently executing process with pID equal to i. The function $NextLabel(x)$ returns the label of the next statement in the execution of the process. 'v and v', where v is a variable, represent the value of that variable before and after P/V operation, respectively. Figure A.2 contain the tabular specification of $V(sem)$ operation.

	$'sem.cnt > 1$	$'sem.cnt = 1$	$'sem.cnt < 1$
$sem.cnt'\|$	$false$	$sem.cnt' = `sem.cnt - 1$	$sem.cnt' = `sem.cnt - 1$
$sem.set'\|$	$false$	$sem.cnt' = `sem.set$	$sem.set' = `sem.cnt \cup \{i\}$
$next'\|$	$false$	Table a)	Table b)

Table a): $\forall j$,

$j = i$	$j \neq i$
$next[j]' = NextLabel(x)$	$next[j]' =' next[j]$

Table b): $\forall j$,

$j = i$	$j \neq i$
$next[j]' = waitAtPsem$	$next[j]' = `next[j]$

Figure A.1: Specification of $P(sem)$ operation

	$'sem.cnt > 0$	$`sem.cnt = 0$	$`sem.cnt < 0$
$sem.cnt'\|$	$false$	$sem.cnt' = `sem.cnt + 1$	$sem.cnt' = `sem.cnt + 1$
$sem.set'\|$	$false$	$sem.cnt' = `sem.set$	$\exists t : (t \in `sem.set \wedge sem.set' = `sem.set - \{t\})$
$next'\|$	$false$	Table a)	Table b)

Table a): $\forall j$,

$j = i$	$j \neq i$
$next[j]' = NextLabel(x)$	$next[j]' = `next[j]$

Table b): $\forall j$,

$j = i$	$j \neq i \wedge j \in (`sem.set - sem.set') \wedge `next[j] = waitAtPsem$	$j \neq i \wedge \neg(j \in (`sem.set - sem.set') \wedge `next[j] = waitAtPsem)$
$next[j]' = NextLabel(x)$	$next[j]' = rlseAtPsem$	$next[j]' = `next[j]$

Figure A.2: Specification of V(sem) operation

Appendix B

The Tabular Representation of the Rewritten Readers/Writers Program

The tabular representation of the rewritten program as given in [15] is given in figures B.1 and B.2.

	$pID = k \wedge IsReader$											
	$`next[k] = r1$			$`next[k] = waitAtPm1$	$`next[k] = rlseAtPm1$	$next[k] = r2$	$`next[k] = r3$					
							$`rdcnt = 1 \wedge$			$`rdcnt < 1$	$`rdcnt > 1 \wedge$	
	$`m.cnt > 1$	$`m.cnt = 1$	$`m.cnt < 1$				$`w.cnt > 1$	$`w.cnt = 1$	$`w.cnt < 1$		$`w.cnt < 1$	$`w.cnt \geq 1$
$rdcnt' =$		$`rdcnt$	$`rdcnt$	$`rdcnt$	$`rdcnt$	$rdcnt + 1$		$`rdcnt$	$`rdcnt$		$`rdcnt$	
$rd' =$		$`rd$	$`rd$	$`rd$	$`rd$	$`rd$		$`rd + 1$	$`rd$		$`rd + 1$	
$wt' =$		$`wt$	$`wt$	$`wt$	$`wt$	$`wt$		$`wt$	$`wt$		$`wt$	
$m.cnt' =$		$`m.cnt - 1$	$`m.cnt - 1$	$`m.cnt$	$`m.cnt$	$`m.cnt$		$`m.cnt$	$`m.cnt$		$`m.cnt$	
$m.set'\|$		$m.set' = `m.set$	$m.set' = `m.set \cup \{k\}$	$m.set' = `m.set$	$m.set' = `m.set$	$m.set' = `m.set$		$m.set' = `m.set$	$m.set' = `m.set$		$m.set' = `m.set$	
$w.cnt' =$		$`w.cnt$	$`w.cnt$	$`w.cnt$	$`w.cnt$	$`w.cnt$		$`w.cnt - 1$	$`w.cnt - 1$		$`w.cnt$	
$w.set'\|$		$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$		$w.set' = `w.set$	$w.set' = `w.set \cup \{k\}$		$w.set' = `w.set$	
$pID'\|$		$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP$	
$next'\|$		Tab2	Tab3	Tab4	Tab5	Tab6		Tab8	Tab9		Tab11	
	1	2	3	4	5	6	7	8	9	10	11	12

	$pID = k \wedge IsReader$										
	$`next[k] = waitAtPwr$	$`next[k] = rlseAtPwr$	$`next[k] = r4$			$`next[k] = r5$	$`next[k] = r6$			$`next[k] = waitAtPm2$	$`next[k] = rlseAtPm2$
			$`m.cnt > 0$	$`m.cnt = 0$	$`m.cnt < 0$		$`m.cnt > 1$	$`m.cnt = 1$	$`m.cnt < 1$		
$rdcnt' =$	$`rdcnt$	$`rdcnt$		$`rdcnt$	$`rdcnt$	$`rdcnt$		$`rdcnt$	$`rdcnt$	$`rdcnt$	$`rdcnt$
$rd' =$	$`rd$	$`rd + 1$		$`rd$	$`rd$	$`rd$		$`rd$	$`rd$	$`rd$	$`rd$
$wt' =$	$`wt$	$`wt$		$`wt$	$`wt$	$`wt$		$`wt$	$`wt$	$`wt$	$`wt$
$m.cnt' =$	$`m.cnt$	$`m.cnt$		$`m.cnt + 1$	$`m.cnt+1$	$`m.cnt$		$`m.cnt - 1$	$`m.cnt - 1$	$`m.cnt$	$`m.cnt$
$m.set'\|$	$m.set' = `m.set$	$m.set' = `m.set$		$m.set' = `m.set$	$\exists t : (t \in `m.set \wedge m.set' = `m.set - \{t\})$	$m.set' = `m.set$		$m.set' = `m.set$	$m.set' = `m.set \cup \{k\}$	$m.set' = `m.set$	$m.set' = `m.set$
$w.cnt' =$	$`w.cnt$	$`w.cnt$		$`w.cnt$	$`w.cnt$	$`w.cnt$		$`w.cnt$	$`w.cnt$	$`w.cnt$	$`w.cnt$
$w.set'\|$	$w.set' = `w.set$	$w.set' = `w.set$		$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$		$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$
$pID'\|$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$
$next'\|$	Tab13	Tab14		Tab16	Tab17	Tab18		Tab20	Tab21	Tab22	Tab23
	13	14	15	16	17	18	19	20	21	22	23

Figure B.1: The tabular representation of the rewritten Readers/Writers Program

	$pID = k \wedge IsReader$								
	$`next[k] = r7$	$`next[k] = r8$					$`next[k] = r9$		
		$`rdcnt = 0$			$`rdcnt < 0$	$`rdcnt > 0$	$`m.cnt > 0$	$`m.cnt = 0$	$`m.cnt < 0$
		$`w.cnt > 0$	$`w.cnt = 0$	$`w.cnt < 0$					
$rdcnt' =$	$`rdcnt - 1$		$`rdcnt$	$`rdcnt$		$`rdcnt$		$`rdcnt$	$`rdcnt$
$rd' =$	$`rd$		$`rd - 1$	$`rd - 1$		$`rd - 1$		$`rd$	$`rd$
$wt' =$	$`wt$		$`wt$	$`wt$		$`wt$		$`wt$	$`wt$
$m.cnt' =$	$`m.cnt$		$`m.cnt$	$`m.cnt$		$`m.cnt$		$`m.cnt + 1$	$`m.cnt + 1$
$m.set' \mid$	$m.set' = `m.set$		$m.set' = `m.set$	$m.set' = `m.set$		$m.set' = `m.set$		$m.set' = `m.set$	$\exists t : (t \in `m.set \wedge m.set' = `m.set - \{t\})$
$w.cnt' =$	$`w.cnt$		$`w.cnt + 1$	$`w.cnt + 1$		$`w.cnt$		$`w.cnt$	$`w.cnt$
$w.set' \mid$	$w.set' = `w.set$		$w.set' = `w.set$	$\exists t : (t \in `m.set \wedge m.set' = `m.set - \{t\})$		$w.set' = `w.set$		$w.set' = `w.set$	$w.set' = `w.set$
$pID' \mid$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP \vee \forall i : next'[i] = EOP$	$next'[pID'] \neq EOP$
$next' \mid$	Tab24		Tab26	Tab27		Tab29		Tab31	Tab32
	24	25	26	27	28	29	30	31	32

	$pID = k \wedge IsWriter$								
	$`next[k] = w1$			$next[k] = waitAtPww$	$next[k] = rlseAtPww$	$`next[k] = w2$	$`next[k] = w3$		
	$`w.cnt > 1$	$`w.cnt = 1$	$`w.cnt < 1$				$`w.cnt > 0$	$`w.cnt = 0$	$`w.cnt < 0$
$rdcnt' =$		$`rdcnt$	$`rdcnt$	$`rdcnt$	$`rdcnt$	$`rdcnt$		$`rdcnt$	$`rdcnt$
$rd' =$		$`rd$	$`rd$	$`rd$	$`rd$	$`rd$		$`rd$	$`rd$
$wt' =$		$`wt + 1$	$`wt$	$`wt$	$`wt + 1$	$`wt$		$`wt - 1$	$`wt - 1$
$m.cnt' =$		$`m.cnt$	$`m.cnt$	$`m.cnt$	$`m.cnt$	$`m.cnt$		$`m.cnt$	$`m.cnt$
$m.set' \mid$		$m.set' = `m.set$	$m.set' = `m.set$	$m.set' = `m.set$	$m.set' = `m.set$	$m.set' = `m.set$		$m.set' = `m.set$	$m.set' = `m.set$
$w.cnt' =$		$`w.cnt - 1$	$`w.cnt - 1$	$`w.cnt$	$`w.cnt$	$`w.cnt$		$`w.cnt + 1$	$`w.cnt + 1$
$w.set' \mid$		$w.set' = `w.set$	$w.set' = `w.set \cup \{k\}$	$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$		$w.set' = `w.set$	$\exists t : (t \in `w.set \wedge w.set' = `w.set - \{t\})$
$pID' \mid$		$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$	$next'[pID'] \neq EOP$		$next'[pID'] \neq EOP \vee \forall i : next'[i] = EOP$	$next'[pID'] \neq EOP$
$next' \mid$		Tab34	Tab35	Tab36	Tab37	Tab38		Tab40	Tab41
	1	2	3	4	5	6	7	8	9

Figure B.2: Figure B.1 continued

Tab2: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r2$	$next[j]' =' next[j]$

Tab3: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = waitAtPm1$	$next[j]' =' next[j]$

Tab4: $\forall j$,

$j = k$	$j \neq k$
$next[j]' =' next[j]$	$next[j]' =' next[j]$

Tab5: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r2$	$next[j]' =' next[j]$

Tab6: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r3$	$next[j]' =' next[j]$

Tab8: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r4$	$next[j]' =' next[j]$

Tab9: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = waitAtPwr$	$next[j]' =' next[j]$

Tab11: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r4$	$next[j]' =' next[j]$

Tab13: $\forall j$,

$j = k$	$j \neq k$
$next[j]' =' next[j]$	$next[j]' =' next[j]$

Tab14: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r4$	$next[j]' =' next[j]$

Tab16: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r5$	$next[j]' =' next[j]$

Tab17: $\forall j$,

$j = k$	$j \neq k \wedge j \in ('m.set - m.set')$		$j \neq k \wedge (j \notin ('m.set - m.set') \vee \neg('next[j] = waitAtPm1 \vee' next[j] = waitAtPm2))$
	$'next[j] = waitAtPm1$	$'next[j] = waitAtPm2$	
$next[j]' = r5$	$next[j]' = rlseAtPm1$	$next[j]' = rlseAtPm2$	$next[j]' =' next[j]$

Tab18: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r6$	$next[j]' =' next[j]$

Tab20: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r7$	$next[j]' =' next[j]$

Tab21: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = waitAtPm2$	$next[j]' =' next[j]$

Tab22: $\forall j$,

$j = k$	$j \neq k$
$next[j]' =' next[j]$	$next[j]' =' next[j]$

Tab23: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r7$	$next[j]' =' next[j]$

Tab24: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r8$	$next[j]' =' next[j]$

Tab26: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r9$	$next[j]' =' next[j]$

Tab27: $\forall j$,

$j = k$	$j \neq k \wedge j \in ('m.set - m.set')$ $'next[j] = waitAtPww$	$j \neq k \wedge (j \notin ('m.set - m.set')\vee$ $\neg' next[j] = waitAtPww)$
$next[j]' = r9$	$next[j]' = rlseAtPww$	$next[j]' =' next[j]$

Tab29: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = r9$	$next[j]' =' next[j]$

Tab31: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = EOP$	$next[j]' =' next[j]$

Tab32: $\forall j$,

$j = k$	$j \neq k \wedge j \in ('m.set - m.set')$		$j \neq k \wedge (j \notin ('m.set - m.set')\vee$ $\neg('next[j] = waitAtPm1 \vee' next[j] = waitAtPm2))$
	$'next[j] = waitAtPm1$	$'next[j] = waitAtPm2$	
$next[j]' = EOP$	$next[j]' = rlseAtPm1$	$next[j]' = rlseAtPm2$	$next[j]' =' next[j]$

Tab34: $\forall j$,

$j = k$	j$\neq k$
$next[j]' = w2$	$next[j]' =' next[j]$

Tab35: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = waitAtPww$	$next[j]' =' next[j]$

Tab36: $\forall j$,

$j = k$	$j \neq k$
$next[j]' =' next[j]$	$next[j]' =' next[j]$

Tab37: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = w2$	$next[j]' =' next[j]$

Tab38: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = w3$	$next[j]' =' next[j]$

Tab40: $\forall j$,

$j = k$	$j \neq k$
$next[j]' = EOP$	$next[j]' =' next[j]$

Tab41: $\forall j$,

$j = k$	$j \neq k \wedge j \in ('w.set - w.set')$		$j \neq k \wedge (j \notin ('w.set - w.set')\vee$ $\neg('next[j] = waitAtPww \vee' next[j] = waitAtPwr))$
	$'next[j] = waitAtPww$	$'next[j] = waitAtPwr$	
$next[j]' = EOP$	$next[j]' = rlseAtPww$	$next[j]' = rlseAtPwr$	$next[j]' =' next[j]$

Appendix C

The Readers/Writers Model in SPIN, SAL, and PVS

C.1 The Readers/Writers Model in SPIN

```
mtype {p, v};
chan mutex = [0] of {mtype};
chan w = [0] of {mtype};
int wt, rd, rdcnt = 0;
active proctype m1()
{
        byte count=1;
        do
        :: (count == 1) ->
end:    mutex!p; count = 0
        :: (count == 0) ->
        mutex?v; count = 1
od
}
active proctype m2()
{
        byte count=1;
        do
        :: (count == 1) ->
end:            w!p; count = 0
        :: (count == 0) ->
                w?v; count = 1
        od
}
active [10] proctype reader()
{
           mutex?p;
           rdcnt++;
           atomic{
           if
           :: rdcnt == 1 ->
              w?p;
           :: else ->
           fi;
           rd++}
           mutex!v;
```

```
 critical: skip;
           mutex?p;
           rdcnt--;
           atomic
           {
            rd--;
            if
            :: rdcnt == 0 -> w!v
            :: else ->
            fi
           };
           mutex!v;
}
active [10] proctype writer()
{
          atomic{
                 w?p; wt++
                };
critical: skip;
          atomic{
                 wt--; w!v
                };
}
```

C.2 Model of Readers/Writers Program in SAL

```
rwf3{; M : nznat, n : nat}: CONTEXT =
BEGIN
  Job_Idx: TYPE = [1..M];
  label: TYPE = {r1, waitAtPm1, rlseAtPm1, r2, r3, waitAtPwr, rlseAtPwr,
r4, r5, r6, waitAtPm2, rlseAtPm2, r7, r8, r9, w1, w2, w3,
waitAtPww, rlseAtPww, EOP};
  rdtype: TYPE = [-1..n+1];
  wttype: TYPE = [-1..(M-n+1)];
  semtype: TYPE = [-M..2];
  index: TYPE = [1..M];
  setof: TYPE = [index -> bool];
  member(x: index, a: setof): bool = a(x);
  empty?(a: setof): bool = (FORALL (x: index): NOT member(x, a));
  emptysetof: setof = {x: index | false};
  union(a: setof, b: setof): setof = {x: index | member(x, a) OR member(x, b)};
  remove(x: index, a: setof): setof = {y: index | x /= y AND member(y, a)};
  sem: TYPE = [#cnt: semtype,
set: setof #];
  IntRW(x: label): int =
         IF  x=r1   THEN     15
         ELSIF x=waitAtPm1 THEN         14
         ELSIF x=rlseAtPm1 THEN 13
         ELSIF x=r2   THEN      12
         ELSIF x=r3   THEN      11
         ELSIF x=waitAtPwr THEN 10
         ELSIF x=rlseAtPwr THEN 9
         ELSIF x=r4 THEN                8
         ELSIF x=r5  THEN               7
         ELSIF x=r6 THEN                6
         ELSIF x=waitAtPm2 THEN 5
         ELSIF x=rlseAtPm2 THEN 4
         ELSIF x=r7   THEN      3
         ELSIF x=r8    THEN     2
         ELSIF x=r9   THEN     1
         ELSIF x=w1   THEN      5
         ELSIF x=waitAtPww THEN 4
         ELSIF x=rlseAtPww THEN 3
         ELSIF x=w2   THEN      2
```

```
        ELSIF x=w3   THEN      1
        ELSIF x=EOP  THEN      0
        ELSE 0
   ENDIF;
  state: TYPE = [#
m: sem,
w: sem,
rdcnt: rdtype,
next: ARRAY index OF label,
rd: rdtype,
wt: wttype #];
  DQdecrease(s, t: state): bool = (EXISTS (i: index):
                  IntRW(t.next[i]) < IntRW(s.next[i])) AND
                  FORALL (i: index):
                  (IntRW(t.next[i]) <= IntRW(s.next[i]));
  process [pID : index]: MODULE =
    BEGIN
      GLOBAL s : state
      LOCAL IsReader : bool
      INITIALIZATION
        IsReader = IF (pID <= n)
THEN  TRUE
ELSE  FALSE ENDIF;
        TRANSITION
        [
         c1:
           IsReader AND s.next[pID] = r1 AND  s.m.cnt > 1
           -->
         []
         c2:
          IsReader AND s.next[pID] = r1 AND s.m.cnt = 1
          --> s' = ((s WITH  .m.cnt := s.m.cnt - 1) WITH .next[pID] := r2)
    []
 c3:
   IsReader AND s.next[pID] = r1 AND s.m.cnt < 1
          --> s' = ((s WITH .m.cnt := s.m.cnt - 1) WITH
.m.set := union({x: index | x = pID}, s.m.set))
    WITH .next[pID] := waitAtPm1
 []
 c4:
   IsReader AND s.next[pID] = waitAtPm1
          --> s' = s
 []
 c5:
   IsReader AND s.next[pID] = rlseAtPm1
          --> s' = (s WITH .next[pID] := r2)
 []
 c6:
   IsReader AND s.next[pID] = r2
          --> s' = (s WITH .rdcnt := s.rdcnt + 1)
       WITH  .next[pID] := r3
         []
 c7:
   IsReader AND s.next[pID] = r3 AND s.rdcnt = 1 AND s.w.cnt > 1
          -->
 []
 c8:
   IsReader AND s.next[pID] = r3 AND s.rdcnt = 1 AND s.w.cnt = 1
          --> s' = ((s WITH .rd := s.rd + 1) WITH
.w.cnt := s.w.cnt - 1 )
    WITH .next[pID] := r4
         []
 C9:
   IsReader AND s.next[pID] = r3 AND s.rdcnt = 1 AND s.w.cnt < 1
          -->  s' = ((s WITH .w.cnt := s.w.cnt - 1) WITH
.w.set := union({x: index | x = pID}, s.w.set))
    WITH .next[pID] := waitAtPwr
 []
```

```
c10:
  IsReader AND s.next[pID] = r3 AND s.rdcnt < 1
         -->
[]
c11:
  IsReader AND s.next[pID] = r3 AND s.rdcnt > 1 AND s.w.cnt < 1
         -->  s' = (s WITH .rd := s.rd + 1)
   WITH .next[pID] := r4
[]
c12:
  IsReader AND s.next[pID] = r3 AND s.rdcnt > 1 AND s.w.cnt >= 1
         -->
[]
c13:
  IsReader AND s.next[pID] = waitAtPwr
         --> s' = s
[]
c14:
  IsReader AND s.next[pID] = rlseAtPwr
         -->  s' = (s WITH .rd := s.rd + 1)
   WITH .next[pID] := r4
[]
c15:
  IsReader AND s.next[pID] = r4 AND s.m.cnt > 0
         -->
[]
c16:
  IsReader AND s.next[pID] = r4 AND s.m.cnt = 0
         --> s' = (s WITH .m.cnt := s.m.cnt + 1)
      WITH  .next[pID] := r5
[]
([] (p: index):
        %c17:
          IsReader AND s.next[pID] = r4  AND
                        s.m.cnt < 0 AND s.m.set(p)
         -->  s' = (((s WITH .m.cnt := s.m.cnt + 1)
      WITH .next[pID] := r5)
   WITH .next[p] :=
                        IF s.next[p] = waitAtPm1 THEN rlseAtPm1
                        ELSE rlseAtPm2
                        ENDIF)
     WITH .m.set := remove(p, s.m.set))
[]
c18:
  IsReader AND s.next[pID]= r5
         --> s' = (s WITH .next[pID] := r6)
[]
c19:
  IsReader AND s.next[pID] = r6 AND s.m.cnt > 1
         -->
[]
c20:
  IsReader AND s.next[pID] = r6 AND s.m.cnt = 1
         -->  s' = (s WITH .m.cnt := s.m.cnt - 1)
   WITH .next[pID] := r7
[]
c21:
  IsReader AND s.next[pID] = r6 AND s.m.cnt < 1
         -->  s' = ((s WITH .m.cnt := s.m.cnt - 1) WITH
.m.set := union({x: index | x = pID}, s.m.set))
   WITH .next[pID] := waitAtPm2
[]
c22:
  IsReader AND s.next[pID] = waitAtPm2
         --> s' = s
[]
c23:
  IsReader AND s.next[pID] = rlseAtPm2
```

```
        -->  s' = s WITH .next[pID] := r7
[]
c24:
  IsReader AND s.next[pID] = r7
        -->  s' = (s WITH .rdcnt := s.rdcnt - 1)
   WITH .next[pID] := r8
[]
c25:
  IsReader AND s.next[pID] = r8 AND s.rdcnt = 0 AND s.w.cnt > 0
        -->
[]
c26:
  IsReader AND s.next[pID] = r8 AND s.rdcnt = 0 AND s.w.cnt = 0
        --> s' = ((s WITH .rd := s.rd - 1)
WITH .w.cnt := s.w.cnt + 1)
   WITH .next[pID] := r9
[]
([] (p: index):
       c27:
         IsReader AND s.next[pID] = r8 AND s.rdcnt = 0
  AND s.w.cnt < 0 AND s.w.set(p) AND  s.next[p] = waitAtPww
        --> s' =
((((s WITH .rd := s.rd - 1)
   WITH .next[pID] := r9)
WITH .w.cnt := s.w.cnt + 1)
WITH .next[p] := rlseAtPww)
WITH .w.set := remove(p, s.w.set))
[]
c28:
  IsReader AND s.next[pID] = r8 AND s.rdcnt < 0
        -->
[]
c29:
  IsReader AND s.next[pID] = r8 AND s.rdcnt > 0
        -->  s' = (s WITH .rd := s.rd - 1)
   WITH .next[pID] := r9
[]
c30:
  IsReader AND s.next[pID] = r9 AND s.m.cnt > 0
        -->
[]
c31:
  IsReader AND s.next[pID] = r9 AND s.m.cnt = 0
        -->  s' = (s WITH .m.cnt := s.m.cnt + 1)
   WITH .next[pID] := EOP
[]
([] (p: index):
       c32:
         IsReader AND s.next[pID] = r9  AND
                        s.m.cnt < 0 AND s.m.set(p)
        -->  s' =
(((s WITH .m.cnt := s.m.cnt + 1)
   WITH .next[pID] := EOP)
WITH .next[p] :=
                         IF s.next[p] = waitAtPm1 THEN rlseAtPm1
                         ELSE rlseAtPm2
          ENDIF)
               WITH .m.set := remove(p, s.m.set))
[]
c33:
  NOT IsReader AND s.next[pID] = w1 AND s.w.cnt > 1
        -->
[]
c34:
  NOT IsReader AND s.next[pID] = w1 AND s.w.cnt = 1
        -->  s' = ((s WITH .wt := s.wt + 1) WITH
.w.cnt := s.w.cnt - 1)
    WITH .next[pID] := w2
```

```
 []
 c35:
   NOT IsReader AND s.next[pID] = w1 AND s.w.cnt < 1
         -->  s' = ((s WITH .w.cnt := s.w.cnt - 1) WITH
.w.set := union({x: index | x = pID}, s.w.set))
WITH .next[pID] := waitAtPww
 []
 c36:
   NOT IsReader AND s.next[pID] = waitAtPww
         --> s' = s
 []
 c37:
   NOT IsReader AND s.next[pID] = rlseAtPww
         -->  s' = (s WITH .wt := s.wt + 1)
WITH .next[pID] := w2
 []
 c38:
   NOT IsReader AND s.next[pID] = w2
         -->  s' = s WITH .next[pID] := w3
 []
 c39:
   NOT IsReader AND s.next[pID] = w3 AND s.w.cnt > 0
         -->
 []
 c40:
   NOT IsReader AND s.next[pID] = w3 AND s.w.cnt = 0
         -->  s' = ((s WITH .wt := s.wt - 1) WITH
.w.cnt := s.w.cnt + 1)
    WITH .next[pID] := EOP
 []
 ([] (p: index):
         c41:
           NOT IsReader AND s.next[pID] = w3 AND
                          s.w.cnt < 0 AND  s.w.set(p)
          --> s' =
 ((((s WITH .wt := s.wt - 1)
    WITH .next[pID] := EOP)
 WITH .w.cnt := s.w.cnt + 1)
 WITH .next[p] :=
                IF s.next[p] = waitAtPww THEN  rlseAtPww
 ELSE  rlseAtPwr
 ENDIF)
 WITH .w.set := remove(p, s.w.set))
 ]
    END;
   dqmonitor : MODULE =
    BEGIN
      INPUT s : state
      OUTPUT prev_state : state
      INITIALIZATION
      prev_state = ((# m := (# cnt := 1, set := {x:index | false} #),
w := (# cnt := 1, set := {x: index | false} #),
               rdcnt := 0, next := [[i:index] IF i <= n THEN r1
                 ELSE w1
              ENDIF],
rd := 0, wt := 0 #))
      TRANSITION
       prev_state' = s;
    END;
   initializator: MODULE =
    BEGIN
      GLOBAL s: state
      INITIALIZATION
        s = ((# m := (# cnt := 1, set := {x:index | false} #),
w := (# cnt := 1, set := {x: index | false} #),
               rdcnt := 0, next := [[i:index] IF i <= n THEN r1
                 ELSE w1
              ENDIF],
```

```
rd := 0, wt := 0 #))
      TRANSITION
       [
             FORALL (i: index): s.next[i] = EOP
            --> s' = s
       ]
    END;
   main: MODULE = initializator []
     ([] (pID : index): process[pID]);
```

C.3 PVS files

1. decl.pvs

```
decl: THEORY

  BEGIN
  %M is the number of processes, n is the number of readers
  M: posnat
  ntype: TYPE = {i: nat | i <= M}
  index: TYPE = {i: ntype | i >= 1} CONTAINING 1
  n: ntype
  label: TYPE = {r1, waitAtPm1, rlseAtPm1, r2, r3, waitAtPwr,
                 rlseAtPwr, r4, r5, r6, waitAtPm2, rlseAtPm2,
                 r7, r8, r9, w1, w2, w3,
waitAtPww, rlseAtPww, EOP}
  x: VAR label
  rlabel?(x): bool = (x = r1 or x =  waitAtPm1 or
                              x = rlseAtPm1 or  x = r2 or
                              x = r3 or x = waitAtPwr or
                              x = rlseAtPwr or x = r4 or
                              x = r5 or x = r6 or
                              x = waitAtPm2 or x = rlseAtPm2 or
                              x = r7 or x = r8 or
                              x = r9 or x = EOP)
  wlabel?(x): bool = (x = w1 or x = w2 or x = w3 or
                              x =  waitAtPww or x = rlseAtPww or
                              x = EOP)
  %we use finite sets, because we'll need to play with cardinality
  %in order to prove safety and clean completion
  importing finite_sets[index]
  ar: TYPE = {a: [index -> label] | forall (i: index):
                      ((i <= n =>  rlabel?(a(i))) and
                       (i > n => wlabel?(a(i))))}
  IsReader(i: index): bool = (i <= n)
  IntRW(x): int =
  COND
 x=r1  ->15,
 x=waitAtPm1 ->  14,
 x=rlseAtPm1 ->13,
   x=r2 ->12,
 x=r3 ->11,
 x=waitAtPwr ->10,
 x=rlseAtPwr ->9,
 x=r4 ->8,
 x=r5  ->7,
 x=r6 ->6,
 x=waitAtPm2 ->5,
 x=rlseAtPm2 ->4,
 x=r7 ->      3,
 x=r8  ->2,
 x=r9 ->1,
 x=w1 ->5,
 x=waitAtPww ->4,
```

```
x=rlseAtPww ->3,
x=w2 ->2,
x=w3 ->1,
x=EOP ->0

 ENDCOND
 sem: TYPE = [#cnt: integer, set: finite_set#]
 state: TYPE = [#
pID: index,
               m: sem,
w: sem,
rdcnt: int,
next: ar,
rd: int,
wt: int  #]
 %we need stateneop type to make sure that the next chosen cannot
 %be the process who finished executing
 stateneop: TYPE = {s: state | next(s)(pID(s)) /= EOP}
 END decl
```

2. conds.pvs

```
conds: THEORY

  BEGIN
  importing decl
  s: VAR stateneop
  p1(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r1 and cnt(m(s)) > 1
  p2(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r1 and cnt(m(s)) = 1
  p3(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r1 and cnt(m(s)) < 1
  p4(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = waitAtPm1
  p5(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = rlseAtPm1
  p6(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r2
  p7(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r3 and
                  rdcnt(s) = 1 and cnt(w(s)) > 1
  p8(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r3 and
                  rdcnt(s) = 1 and cnt(w(s)) = 1
  p9(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r3 and
                  rdcnt(s) = 1 and cnt(w(s)) < 1
  p10(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r3 and rdcnt(s) < 1
  p11(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r3 and
                   rdcnt(s) > 1 and cnt(w(s)) < 1
  p12(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r3 and
                   rdcnt(s) > 1 and cnt(w(s)) >= 1
  p13(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = waitAtPwr
  p14(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = rlseAtPwr
  p15(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r4 and cnt(m(s)) >0
  p16(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r4 and cnt(m(s)) =  0
  p17(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r4 and cnt(m(s)) < 0
  p18(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r5
  p19(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r6 and cnt(m(s)) > 1
  p20(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r6 and cnt(m(s)) = 1
  p21(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r6 and cnt(m(s)) < 1
  p22(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = waitAtPm2
  p23(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = rlseAtPm2
  p24(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r7
  p25(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r8 and
                   rdcnt(s) = 0 and cnt(w(s)) > 0
  p26(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r8 and
                   rdcnt(s) = 0 and cnt(w(s)) = 0
  p27(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r8 and
                   rdcnt(s) = 0 and cnt(w(s)) < 0
```

```
  p28(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r8 and rdcnt(s) < 0
  p29(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r8 and rdcnt(s) > 0
  p30(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r9  and cnt(m(s)) > 0
  p31(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r9  and cnt(m(s)) = 0
  p32(s): bool = IsReader(pID(s)) and next(s)(pID(s)) = r9  and cnt(m(s)) < 0
  p33(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w1  and cnt(w(s)) > 1
  p34(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w1  and cnt(w(s)) = 1
  p35(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w1  and cnt(w(s)) < 1
  p36(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = waitAtPww
  p37(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = rlseAtPww
  p38(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w2
  p39(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w3  and cnt(w(s)) > 0
  p40(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w3  and cnt(w(s)) = 0
  p41(s): bool = not IsReader(pID(s)) and next(s)(pID(s)) = w3  and cnt(w(s)) < 0

  END conds
```

3. table.pvs

```
transition  % [ parameters ]
: THEORY

  BEGIN
  importing conds
  j: VAR index
  trans(s : {s:stateneop |
        NOT (p1(s) or p7(s) or p10(s) or p12(s)
        or p15(s) or p19(s) or p25(s) or p28(s)
        or p30(s) or p33(s) or p39(s))}, t: state): bool =
  LET k: index = pID(s) IN
  table
  %-------------------------------------------------------------||
  |p1(s)|                                                       ||
  %-------------------------------------------------------------||
  |p2(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
          wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) - 1 and       %
          set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and   %
          set(w(t)) = set(w(s)) and                             %
          (forall j: (j= k and next(t)(j) = r2) or              %
          (j /= k and next(t)(j) = next(s)(j))) and             %
          next(t)(pID(t)) /= EOP                                ||
  %-------------------------------------------------------------||
  | p3(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and            %
           wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) - 1 and      %
           set(m(t)) =  add (k, set(m(s))) and                  %
           cnt(w(t)) = cnt(w(s)) and set(w(t)) = set(w(s)) and  %
           (forall j: (j= k and next(t)(j) = waitAtPm1) or      %
           (j /= k and next(t)(j) = next(s)(j)))  and           %
           next(t)(pID(t)) /= EOP                               ||
  %-------------------------------------------------------------||
  |p4(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
          wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and           %
          set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and   %
          set(w(t)) = set(w(s)) and                             %
          (forall j: next(t)(j) = next(s)(j))  and        %
          next(t)(pID(t)) /= EOP                                ||
  %-------------------------------------------------------------||
  |p5(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
          wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and           %
          set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and   %
          set(w(t)) = set(w(s)) and                             %
          (forall j: (j= k and next(t)(j) = r2) or        %
          (j /= k and next(t)(j) = next(s)(j)))  and            %
          next(t)(pID(t)) /= EOP                                ||
  %-------------------------------------------------------------||
  |p6(s)| rdcnt(t) = rdcnt(s) + 1 and rd(t) = rd(s) and         %
          wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and           %
```

```
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and     %
        set(w(t)) = set(w(s)) and                               %
        (forall j: (j= k and next(t)(j) = r3) or        %
        (j /= k and next(t)(j) = next(s)(j))) and               %
        next(t)(pID(t)) /= EOP                                  ||
%---------------------------------------------------------------||
|p7(s)|                                                         ||
%---------------------------------------------------------------||
|p8(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) + 1  and          %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and             %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) - 1  and%
        set(w(t)) = set(w(s)) and                               %
        (forall j: (j= k and next(t)(j) = r4) or        %
        (j /= k and next(t)(j) = next(s)(j))) and               %
        next(t)(pID(t)) /= EOP                                  ||
%---------------------------------------------------------------||
|p9(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and               %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and             %
        set(w(t)) = add(k, set(w(s))) and                       %
        cnt(w(t)) = cnt(w(s)) - 1 and set(m(t)) = set(m(s)) and %
        (forall j: (j= k and next(t)(j) = waitAtPwr) or %
        (j /= k and next(t)(j) = next(s)(j))) and               %
        next(t)(pID(t)) /= EOP                                  ||
%---------------------------------------------------------------||
|p10(s)|                                                        ||
%---------------------------------------------------------------||
|p11(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) + 1 and          %
         wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
         set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
         set(w(t)) = set(w(s)) and                              %
         (forall j: (j= k and next(t)(j) = r4) or       %
         (j /= k and next(t)(j) = next(s)(j)))  and             %
         next(t)(pID(t)) /= EOP                                 ||
%---------------------------------------------------------------||
|p12(s)|                                                  ||
%---------------------------------------------------------------||
|p13(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
         wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
         set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
         set(w(t)) = set(w(s)) and                              %
         (forall j: next(t)(j) = next(s)(j)) and        %
         next(t)(pID(t)) /= EOP                                 ||
%---------------------------------------------------------------||
|p14(s)|  rdcnt(t) = rdcnt(s) and rd(t) = rd(s) + 1 and         %
          wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and           %
          set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and   %
          set(w(t)) = set(w(s)) and                             %
          (forall j: (j= k and next(t)(j) = r4) or      %
          (j /= k and next(t)(j) = next(s)(j))) and             %
          next(t)(pID(t)) /= EOP                                ||
%---------------------------------------------------------------||
|p15(s)|                                                        ||
%---------------------------------------------------------------||
|p16(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
         wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) + 1 and        %
         set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
         set(w(t)) = set(w(s)) and                              %
         (forall j: (j= k and next(t)(j) = r5) or       %
         (j /= k and next(t)(j) = next(s)(j))) and              %
         next(t)(pID(t)) /= EOP                                 ||
%---------------------------------------------------------------||
|p17(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
         wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) + 1 and        %
         cnt(w(t)) = cnt(w(s)) and  set(w(t)) = set(w(s)) and   %
         (exists (p:index):(set(m(s))(p) and                    %
         set(m(t)) = remove(p, set(m(s))))  and                 %
         (forall j: (j= k and next(t)(j) = r5) or       %
         (j /= k and difference(set(m(s)), set(m(t)))(j) and    %
```

```
        ((next(s)(j) = waitAtPm1 and next(t)(j) = rlseAtPm1) or%
        (next(s)(j) = waitAtPm2 and next(t)(j) = rlseAtPm2)))  %
        or (j /= k and (not difference(set(m(s)), set(m(t)))(j)%
        or                                                     %
        (next(s)(j) /= waitAtPm1 and next(s)(j) /= waitAtPm2)) %
        and  next(t)(j) = next(s)(j)))  and                    %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p18(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s))  and   %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = r6) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p19(s)|                                                       ||
%-------------------------------------------------------------||
|p20(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) - 1 and        %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = r7) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p21(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) - 1 and        %
        set(m(t)) =  add(k, set(m(s)))  and                    %
        cnt(w(t)) = cnt(w(s)) and  set(w(t)) = set(w(s)) and   %
        (forall j: (j= k and next(t)(j) = waitAtPm2) or%
        (j /= k and next(t)(j) = next(s)(j)))  and             %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p22(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (next(t)(j) = next(s)(j))) and      %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p23(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and              %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = r7) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p24(s)| rdcnt(t) = rdcnt(s) - 1 and rd(t) = rd(s) and          %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = r8) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p25(s)|                                                       ||
%-------------------------------------------------------------||
|p26(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) - 1 and          %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) + 1 and%
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = r9) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p27(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) - 1 and          %
```

```
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        cnt(w(t)) = cnt(w(s)) + 1 and set(m(t)) = set(m(s)) and%
        (exists (p:index):(set(w(s))(p) and                    %
        set(w(t)) = remove(p, set(w(s))))  and                 %
        (forall j: ((j = k and next(t)(k) = r9) or     %
        (j /= k and difference(set(w(s)), set(w(t)))(j)        %
        and next(s)(j) = waitAtPww and next(t)(j) = rlseAtPww) %
        or (j /= k and  (NOT difference(set(w(s)), set(w(t)))(j)%
        or next(s)(j) /= waitAtPww) and                        %
        next(t)(j) = next(s)(j))))  and next(t)(pID(t)) /= EOP ||
%-------------------------------------------------------------||
|p28(s)|                                                      ||
%-------------------------------------------------------------||
|p29(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) - 1 and         %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = r9) or       %
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p30(s)|                                                      ||
%-------------------------------------------------------------||
|p31(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and             %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) + 1  and       %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = EOP) or      %
        (j /= k and next(t)(j) = next(s)(j)))   and            %
        (next(t)(pID(t)) /= EOP or                             %
        forall (i: index): next(t)(i) = EOP)                   ||
%-------------------------------------------------------------||
|p32(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s)  and            %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) + 1  and       %
        cnt(w(t)) = cnt(w(s)) and  set(w(t)) = set(w(s)) and   %
        (exists (p:index):(set(m(s))(p) and                    %
        set(m(t)) =  remove(p, set(m(s)))) and                 %
        (forall j: (j= k and next(t)(j) = EOP) or      %
        (j /= k and difference(set(m(s)), set(m(t)))(j) and    %
        ((next(s)(j) = waitAtPm1 and next(t)(j) = rlseAtPm1) or%
        (next(s)(j) = waitAtPm2 and next(t)(j) = rlseAtPm2))) or%
        (j /= k and (not difference(set(m(s)), set(m(t)))(j)   %
        or (next(s)(j) /= waitAtPm1 and next(s)(j) /= waitAtPm2))%
        and  next(t)(j) = next(s)(j)))  and                     %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p33(s)|                                                      ||
%-------------------------------------------------------------||
|p34(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and             %
        wt(t) = wt(s) + 1 and cnt(m(t)) = cnt(m(s)) and        %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) - 1 and%
        set(w(t)) = set(w(s)) and                              %
        (forall j: (j= k and next(t)(j) = w2) or       %
        (j /= k and next(t)(j) = next(s)(j)))  and             %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p35(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and             %
        wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and            %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) - 1 and%
        set(w(t)) = add(k, set(w(s))) and                      %
        (forall j: (j= k and next(t)(j) = waitAtPww) or%
        (j /= k and next(t)(j) = next(s)(j))) and              %
        next(t)(pID(t)) /= EOP                                 ||
%-------------------------------------------------------------||
|p36(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and             %
        wt(t) = wt(s)  and cnt(m(t)) = cnt(m(s))  and          %
        set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and    %
        set(w(t)) = set(w(s)) and                              %
```

```
         (forall j: next(t)(j) = next(s)(j))  and       %
         next(t)(pID(t)) /= EOP                                  ||
%-------------------------------------------------------------------||
|p37(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and                 %
         wt(t) = wt(s)+ 1 and cnt(m(t)) = cnt(m(s))  and         %
         set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) and     %
         set(w(t)) = set(w(s)) and                               %
         (forall j: (j= k and next(t)(j) = w2) or       %
         (j /= k and next(t)(j) = next(s)(j))) and               %
         next(t)(pID(t)) /= EOP                            ||
%-------------------------------------------------------------------||
|p38(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and                 %
         wt(t) = wt(s) and cnt(m(t)) = cnt(m(s)) and             %
         set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s))  and    %
         set(w(t)) = set(w(s)) and                               %
         (forall j: (j= k and next(t)(j) = w3) or       %
         (j /= k and next(t)(j) = next(s)(j)))  and              %
         next(t)(pID(t)) /= EOP                                  ||
%-------------------------------------------------------------------||
|p39(s)|                                                         ||
%-------------------------------------------------------------------||
|p40(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and                 %
         wt(t) = wt(s) - 1 and cnt(m(t)) = cnt(m(s))  and        %
         set(m(t)) = set(m(s)) and cnt(w(t)) = cnt(w(s)) + 1 and%
         set(w(t)) = set(w(s)) and                               %
         (forall j: (j= k and next(t)(k) = EOP) or      %
         (j /= k and next(t)(j) = next(s)(j)))  and      %
         (next(t)(pID(t)) /= EOP or                              %
         forall (i: index): next(t)(i) = EOP)                    ||
%-------------------------------------------------------------------||
|p41(s)| rdcnt(t) = rdcnt(s) and rd(t) = rd(s) and                 %
         wt(t) = wt(s) - 1 and cnt(m(t)) = cnt(m(s)) and         %
         cnt(w(t)) = cnt(w(s)) + 1 and set(m(t)) = set(m(s)) and%
         (exists (p:index):((set(w(s)))(p) and                   %
         set(w(t)) = remove(p, set(w(s)))))) and                 %
         (forall j: (j= k and next(t)(j) = EOP) or      %
         (j /= k and difference(set(w(s)), set(w(t)))(j) and     %
         ((next(s)(j) = waitAtPww and next(t)(j) = rlseAtPww) or%
         (next(s)(j) = waitAtPwr and next(t)(j) = rlseAtPwr)))  %
         or (j /= k and (not difference(set(w(s)), set(w(t)))(j)%
         or (next(s)(j) /= waitAtPww and next(s)(j) /= waitAtPwr))%
         and  next(t)(j) = next(s)(j))) and                      %
         next(t)(pID(t)) /= EOP                                  ||
endtable
END transition
```

4. getinv.pvs

```
getinv: THEORY

  BEGIN
  importing transition
  s: VAR stateneop
  t: VAR state
  i: VAR index
  rp(t): bool = (wt(t) = 0 or rd(t) = 0) and wt(t) < 2 and rd(t) >= 0
                        and wt(t) >= 0
  initcond(t): bool = cnt(m(t)) = 1 and empty?(set(m(t))) and
                    cnt(w(t)) = 1 and empty?(set(w(t))) and
                    rd(t) = 0 and wt(t) = 0 and rdcnt(t) = 0 and
                    (forall i: (i <= n  and  next(t)(i) = r1)
                      or (i > n  and  next(t)(i) = w1))
  crp11: lemma (forall t:(initcond(t) => rp(t)))
                           and (forall s, t:(rp(s)
                           and trans(s, t) => rp(t)))
   % after the first iteration from failed proofs of crp11
   %we read the invariants as below
```

```
 S7(t): bool = rdcnt(t) >= 0
 S2(t): bool = cnt(w(t)) <=  1
 S1(t): bool = cnt(m(t)) <=  1
 S6(t): bool = cnt(w(t)) = 1 => (wt(t) = 0 and rd(t) = 0)
 S91(t): bool = forall i: next(t)(i) = rlseAtPwr
         implies wt(t) = 0
 S31(t): bool = forall i:
        next(t)(i) = r4 implies cnt(m(t)) <= 0
 S32(t): bool = forall i:
                    next(t)(i) = r9 implies cnt(m(t)) <= 0
 S41(t): bool = forall i: next(t)(i) = r3
         implies rdcnt(t) = rd(t) +  1
 S5(t): bool = forall i: next(t)(i) = r8
                 implies rd(t) = rdcnt(t) + 1
 S81(t): bool = forall i: next(t)(i) = rlseAtPww
                 implies wt(t) = 0
 S82(t): bool = forall i: next(t)(i) = rlseAtPww
                 implies rd(t) = 0
 S101(t): bool = forall i: next(t)(i) = w3
              implies wt(t) = 1
 ind1(t): bool = rp(t) and S1(t) and S2(t) and S31(t)
                 and S32(t) and S41(t) and S5(t)
                 and S6(t) and S7(t) and S81(t)
                 and S82(t) and S91(t) and S101(t)
crpind1: lemma (forall t: initcond(t) => ind1(t))
                          and forall s, t: (ind1(s)
                          and trans(s, t) => ind1(t)) %2619
% from the unprovable sequents we got more
         %invariants, which together with previous ones, give us new set:
 S92(t): bool = forall i: next(t)(i) = rlseAtPwr
         implies cnt(w(t)) <= 0
 S35(t): bool = forall i:
        next(t)(i) = r3 implies cnt(m(t)) <= 0
 S38(t): bool = forall i:
        next(t)(i) = r8 implies cnt(m(t)) <= 0
 S39(t): bool = forall i:
        next(t)(i) = rlseAtPwr implies cnt(m(t)) <= 0
 S10(t): bool = forall i: next(t)(i) = w2
                 implies wt(t) = 1
 S83(t): bool = forall i: next(t)(i) = rlseAtPww
                 implies cnt(w(t)) <= 0
 S111(t): bool = forall i: next(t)(i) = r7
                 implies rd(t) = rdcnt(t)
 S112(t): bool = forall i: next(t)(i) = r2
                 implies rd(t) = rdcnt(t)
 S125(t): bool = forall i:  next(t)(i) = r7
                 implies rd(t) >= 1
 ind2(t): bool = ind1(t) and S92(t) and S35(t) and
                 S38(t) and S39(t) and S10(t) and S83(t) and S111(t) and
                 S112(t) and S125(t)
 ind2a(t): bool =  S31(t)
                 and S32(t) and  S5(t)
                 and S6(t) and S7(t) and S91(t)
                 and S101(t) and S92(t) and S35(t) and
                 S38(t) and S39(t) and S10(t) and S83(t) and S111(t) and
                 S112(t) and S125(t)
crpind2: lemma (forall t: initcond(t) => ind2a(t))
                          and forall s, t: (ind2(s)
                          and trans(s, t) => ind2a(t)) %new-4577
%new invariants:
 S34(t): bool = forall i:  next(t)(i) = r2
                      implies cnt(m(t)) <= 0
 S37(t): bool = forall i:  next(t)(i) = r7
                      implies cnt(m(t)) <= 0
 S114(t): bool = forall i: next(t)(i) = rlseAtPm2
                 implies rd(t) = rdcnt(t)
 S115(t): bool = forall i: next(t)(i) = rlseAtPm1
                      implies rd(t) = rdcnt(t)
```

```
  S131(t): bool = forall i:
                next(t)(i)= r1 and cnt(m(t)) = 1
                implies rd(t) = rdcnt(t)
  S123(t): bool = forall i: next(t)(i) = r6
                                implies rd(t) >= 1
  S124(t): bool = forall i: next(t)(i) = rlseAtPm2
                                implies rd(t) >= 1
  S133(t): bool = forall  i: next(t)(i) = r6 and cnt(m(t)) = 1
                                 implies rd(t) = rdcnt(t)
  ind3(t): bool = ind2(t) and S34(t) and S37(t) and S114(t) and
                  S115(t) and S123(t) and S124(t) and S131(t) and S133(t)
  ind3a(t): bool = S35(t) and S38(t) and S111(t) and S112(t) and S125(t) and
                  S34(t) and S37(t) and S114(t) and
                  S115(t) and S123(t) and S124(t) and S131(t) and S133(t)
  crpind3: lemma (forall t: initcond(t) => ind3a(t))
                           and forall s, t: (ind3(s)
                           and trans(s, t) => ind3a(t)) %5000
  S33(t): bool = forall i: next(t)(i) = rlseAtPm1
                       implies cnt(m(t)) <= 0
  S36(t): bool = forall i: next(t)(i) = rlseAtPm2
                       implies cnt(m(t)) <= 0
  S122(t): bool = forall i: next(t)(i) = r5
                                implies rd(t) >= 1
  S132(t): bool = forall  i: next(t)(i) = r5 and cnt(m(t)) = 1
                                 implies rd(t) = rdcnt(t)
  ind4(t): bool = ind3(t) and S33(t) and S36(t) and S122(t) and S132(t)
  ind4a(t): bool = S34(t) and S37(t) and S123(t) and S133(t) and
                  S33(t) and S36(t) and S122(t) and S132(t)
  crpind4: lemma (forall t: initcond(t) => ind4a(t))
                           and forall s, t: (ind4(s)
                           and trans(s, t) => ind4a(t)) %3520
  S113(t): bool = forall i: next(t)(i) = r4
                                 implies rd(t) = rdcnt(t)
  S121(t): bool = forall i: next(t)(i) = r4
                implies rd(t) >= 1
  ind5(t): bool = ind4(t) and S113(t) and S121(t)
  ind5a(t): bool = S122(t) and S132(t) and S113(t) and S121(t)
  crpind5: lemma (forall t: initcond(t) => ind5a(t))
                           and forall s, t: (ind5(s)
                           and trans(s, t) => ind5a(t)) %2377
S42(t): bool = forall i: next(t)(i) = rlseAtPwr
                                implies rdcnt(t) = rd(t) + 1
  ind6(t): bool = ind5(t) and S42(t)
  ind6a(t): bool = S113(t) and S42(t)
  crpind6: lemma (forall t: initcond(t) => ind6a(t))
                           and forall s, t: (ind6(s) and
                           trans(s, t) => ind6a(t)) %973
  END getinv
```

5. invj.pvs

```
invj  % [ parameters ]
: THEORY
  % ASSUMING
   % assuming declarations
  % ENDASSUMING



  BEGIN

  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  importing transition
  s: VAR stateneop
```

```
t: VAR state
i, j: VAR index
%safety property:
rp(t): bool = (wt(t) = 0 or rd(t) = 0) and wt(t) < 2 and rd(t) >= 0
                        and wt(t) >= 0
%initial state:
initcond(t): bool = cnt(m(t)) = 1 and empty?(set(m(t))) and
                  cnt(w(t)) = 1 and empty?(set(w(t))) and
                  rd(t) = 0 and wt(t) = 0 and rdcnt(t) = 0 and
                  (forall (i: index): (i <= n  and  next(t)(i) = r1)
                  or (i > n  and  next(t)(i) = w1))
 %invariants as found in inlong.pvs
 S1(t): bool = (cnt(m(t)) <=  1)
 S2(t): bool = (cnt(w(t)) <=  1)
 S6(t): bool = (cnt(w(t)) = 1 => (wt(t) = 0 and rd(t) = 0))
 S7(t): bool = (rdcnt(t) >= 0)
 S31(t): bool = forall i:
        next(t)(i) = r4
           implies cnt(m(t)) <= 0
 S32(t): bool = forall i:
        next(t)(i) = r9
           implies cnt(m(t)) <= 0
 S33(t): bool = forall i:
        next(t)(i) = rlseAtPm1
           implies cnt(m(t)) <= 0 %done, 3535.12
 S34(t): bool = forall i:
        next(t)(i) = r2 implies cnt(m(t)) <= 0% done, 3867
 S35(t): bool = forall i:
        next(t)(i) = r3
           implies cnt(m(t)) <= 0
 S36(t): bool = forall i:
        next(t)(i) = rlseAtPm2
          implies cnt(m(t)) <= 0
 S37(t): bool = forall i:
        next(t)(i) = r7
           implies cnt(m(t)) <= 0
 S38(t): bool = forall i:
        next(t)(i) = r8
           implies cnt(m(t)) <= 0
 S39(t): bool = forall i:
        next(t)(i) = rlseAtPwr
           implies cnt(m(t)) <= 0
 S41(t): bool = forall i: next(t)(i) = r3
         implies rdcnt(t) = rd(t) +  1
 S42(t): bool = forall i:  next(t)(i) = rlseAtPwr
         implies rdcnt(t) = rd(t) +  1
 S5(t): bool = forall i:  next(t)(i) = r8
                implies rd(t) = rdcnt(t) + 1
 S81(t): bool = forall i: next(t)(i) = rlseAtPww
                implies wt(t) = 0
 S82(t): bool = forall i: next(t)(i) = rlseAtPww
                implies rd(t) = 0
 S83(t): bool = forall i: next(t)(i) = rlseAtPww
                implies cnt(w(t)) < 1
 S91(t): bool = forall i: next(t)(i) = rlseAtPwr
         implies wt(t) = 0
 S92(t): bool = forall i: next(t)(i) = rlseAtPwr
         implies cnt(w(t)) <= 0
 S10(t): bool = forall i: next(t)(i) = w2
               implies wt(t) = 1
 S101(t): bool = forall i: next(t)(i) = w3
              implies wt(t) = 1
 S111(t): bool = forall i: next(t)(i) = r7
                implies rd(t) = rdcnt(t)
 S112(t): bool = forall i: next(t)(i) = r2
                implies rd(t) = rdcnt(t)
 S113(t): bool = forall i: next(t)(i) = r4
                implies rd(t) = rdcnt(t)
```

```
 S114(t): bool = forall i: next(t)(i) = rlseAtPm2
                 implies rd(t) = rdcnt(t)
 S115(t): bool = forall i: next(t)(i) = rlseAtPm1
                 implies rd(t) = rdcnt(t)
 S121(t): bool = forall i: next(t)(i) = r4
                 implies rd(t) >= 1
 S122(t): bool = forall i: next(t)(i) = r5
                 implies rd(t) >= 1
 S123(t): bool = forall i: next(t)(i) = r6
                 implies rd(t) >= 1
 S124(t): bool = forall i: next(t)(i) = rlseAtPm2
                 implies rd(t) >= 1
 S125(t): bool = forall i: next(t)(i) = r7
                 implies rd(t) >= 1
 S131(t): bool = forall i: next(t)(i) = r1
                and cnt(m(t)) = 1
               implies rdcnt(t) = rd(t)
 S132(t): bool = forall i: next(t)(i) = r5
                 and cnt(m(t)) = 1
                implies rdcnt(t) = rd(t)
 S133(t): bool = forall i: next(t)(i) = r6
                 and cnt(m(t)) = 1
                implies rdcnt(t) = rd(t)
CS1pred(t, i): bool =  next(t)(i) = rlseAtPm1
              or next(t)(i) = r2 or
              next(t)(i) = r3 or next(t)(i) = r4 or next(t)(i) = rlseAtPm2
              or next(t)(i) = r7 or next(t)(i) = r8
              or next(t)(i) = r9 or next(t)(i) = waitAtPwr or
              next(t)(i) = rlseAtPwr
CS1(t): bool =
              (forall (i, j: index): CS1pred(t, i) and CS1pred(t, j) => i = j)
CS2pred(t, i): bool =  next(t)(i) = w2
              or next(t)(i) = w3 or
              next(t)(i) = rlseAtPwr or next(t)(i) = rlseAtPww
CS2(t): bool =
              (forall (i, j: index): CS2pred(t, i) and CS2pred(t, j)
              => i = j)
indc(t): bool = CS1(t) and CS2(t) and rp(t) and S1(t) and S2(t)
                 and S31(t) and S32(t) and S33(t) and S34(t) and S35(t)
                 and S36(t) and S37(t) and S38(t) and S39(t)
                 and S41(t) and S42(t) and S5(t) and S6(t)
                 and S7(t) and S81(t) and S82(t) and S83(t)
                 and S91(t) and  S92(t) and S10(t) and S101(t)
                 and S111(t) and S112(t) and S113(t) and S114(t) and S115(t)
                 and S121(t) and S122(t) and S123(t) and S124(t) and S125(t)
                 and S131(t) and S132(t) and S133(t)
crpindrp: lemma (forall t: initcond(t) => rp(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => rp(t)) %1464, 1851kipd
crpind1: lemma (forall t: initcond(t) => S1(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => S1(t)) %new-30(s1)
crpind2: lemma (forall t: initcond(t) => S2(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => S2(t)) %new-218(s1)
crpind6: lemma (forall t: initcond(t) => S6(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => S6(t)) %new-4670
crpind7: lemma (forall t: initcond(t) => S7(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => S7(t))%new-130(s1)
crpind31: lemma (forall t: initcond(t) => S31(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => S31(t)) %new-480
crpind32: lemma (forall t: initcond(t) => S32(t))
                           and forall s, t: (indc(s)
                           and trans(s, t) => S32(t)) % new-684(s"r")
crpind33: lemma (forall t: initcond(t) => S33(t))
```

```
                          and forall s, t: (indc(s)
                          and trans(s, t) => S33(t)) %new-298(s"r")
crpind34: lemma (forall t: initcond(t) => S34(t))
                          and forall s, t: (indc(s)
                          and trans(s, t) => S34(t))%new-354(s"r")
crpind35: lemma (forall t: initcond(t) => S35(t))
                          and forall s, t: (indc(s)
                          and trans(s, t) => S35(t)) %new-403
crpind36: lemma (forall t: initcond(t) => S36(t))
                          and forall s, t: (indc(s)
                          and trans(s, t) => S36(t))%new-471(s"r")
crpind37: lemma (forall t: initcond(t) => S37(t))
                          and forall s, t: (indc(s)
                          and trans(s, t) => S37(t)) %new-571
crpind38: lemma (forall t: initcond(t) => S38(t))
                          and forall s, t: (indc(s)
                          and trans(s, t) => S38(t)) %new-612
crpind39: lemma (forall t: initcond(t) => S39(t))
                          and forall s, t: (indc(s)
                          and trans(s, t) => S39(t))
%we found another invariant while proving crpind39:
S140(t): bool = forall i: next(t)(i) = waitAtPwr
              implies cnt(m(t)) <= 0
indc1(t): bool = indc(t) and S140(t)
crpind140: lemma (forall t: initcond(t) => S140(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => S140(t))%new(sr)-644-experiment
crpind39i: lemma (forall t: initcond(t) => S39(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => S39(t))%new-723(s"r"+revinst)
% "divide and conquer" CS1 and CS2, so that proof would be faster
CS11(t): bool =
             forall i, j: next(t)(i) = rlseAtPm1
             and CS1pred(t, j)
             => i = j
CS12(t): bool =
             forall i, j: next(t)(i) = r2
             and CS1pred(t, j)
             => i = j
CS13(t): bool =
             forall i, j: next(t)(i) = r3
             and CS1pred(t, j)
             => i = j
CS14(t): bool =
             forall i, j: next(t)(i) = r4
             and CS1pred(t, j)
             => i = j
CS15(t): bool =
             forall i, j: next(t)(i) = rlseAtPm2
             and CS1pred(t, j)
             => i = j
CS16(t): bool =
             forall i, j: next(t)(i) = r7
             and CS1pred(t, j)
             => i = j
CS17(t): bool =
             forall i, j: next(t)(i) = r8
             and CS1pred(t, j)
             => i = j
CS18(t): bool =
             forall i, j: next(t)(i) = r9
             and CS1pred(t, j)
             => i = j
CS19(t): bool =
             forall i, j: next(t)(i) = waitAtPwr
             and CS1pred(t, j)
             => i = j
CS110(t): bool =
```

```
            forall i, j: next(t)(i) = rlseAtPwr
            and CS1pred(t, j)
            => i = j
 CS21(t): bool =
            forall i, j: next(t)(i) = w2
            and CS2pred(t, j)
            => i = j
CS22(t): bool =
            forall i, j: next(t)(i) = w3
            and CS2pred(t, j)
            => i = j
CS23(t): bool =
            forall i, j: next(t)(i) = rlseAtPww
            and CS2pred(t, j)
            => i = j
CS24(t): bool =
            forall i, j: next(t)(i) = rlseAtPwr
            and CS2pred(t, j)
            => i = j
indcs11: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS11(t))%new-3443
indcs12: lemma (forall t: initcond(t) => CS1(t)) %ne-4026
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS12(t))
indcs13: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s) %ne-3277
                          and trans(s, t) => CS13(t))
indcs14: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS14(t))%ne-3252
indcs15: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS15(t))%ne-3159
indcs16: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS16(t)) %ne-3198css
indcs17: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS17(t)) %ne-3198
indcs18: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS18(t)) %ne-3373(ccs)
indcs19: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS19(t)) %ne-3444(css)
indcs110: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS110(t)) %ne-3505(css)
indcs21: lemma (forall t: initcond(t) => CS2(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS21(t))% ne-3137(css "CS2")
indcs22: lemma (forall t: initcond(t) => CS2(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS22(t)) %ne 3084
indcs23: lemma (forall t: initcond(t) => CS2(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS23(t)) %ne-2940(css+10goals)
indcs24: lemma (forall t: initcond(t) => CS2(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS24(t))%ne-2794(css)
indcs1f: lemma (forall t: initcond(t) => CS1(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS1(t))%new-17
indcs2f: lemma (forall t: initcond(t) => CS2(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => CS2(t))%new-7
crpind41: lemma (forall t: initcond(t) => S41(t))
```

```
                          and forall s, t: (indc1(s)
                          and trans(s, t) => S41(t)) %new-463(s"r")
  crpind42: lemma (forall t: initcond(t) => S42(t))
                          and forall s, t: (indc1(s)
                          and trans(s, t) => S42(t)) %new-705

  % This is where we need S43
   S43(t): bool = forall i: next(t)(i) = waitAtPwr
                implies rdcnt(t) = rd(t) + 1
   indc2(t): bool =  indc1(t) and S43(t)
  crpind43: lemma (forall t: initcond(t) => S43(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S43(t)) %new-654(s"r")
  crpind42i: lemma (forall t: initcond(t) => S42(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S42(t)) %new-726(s"r"and revins)
  crpind5: lemma (forall t: initcond(t) => S5(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S5(t)) %new-690(s"r")
  crpind10: lemma (forall t: initcond(t) => S10(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S10(t)) %new-346(s"w")
  crpind101: lemma (forall t: initcond(t) => S101(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S101(t)) %new-403(s "w")
  crpind81: lemma (forall t: initcond(t) => S81(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S81(t))%new-517,but
                          %we had to use S83 (kindof; s "w")
  crpind82: lemma (forall t: initcond(t) => S82(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S82(t))%new-567(s"w"+s83revins)
  crpind83: lemma (forall t: initcond(t) => S83(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S83(t))%new-406(s "w"+rev)
  crpind91: lemma (forall t: initcond(t) => S91(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S91(t)) %new-512(s "w") rev(S92)
  crpind92: lemma (forall t: initcond(t) => S92(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S92(t))%new-need CS2 too,
              %add after all for one goal and inst-730
  END invj
```

6. invj1.pvs

```
invj1: THEORY

  BEGIN
  importing invj
  s: VAR stateneop
  t: VAR state
  crpind111: lemma (forall t: initcond(t) => S111(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S111(t))%new-584
  crpind112: lemma (forall t: initcond(t) => S112(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S112(t))%new-456
  crpind113: lemma (forall t: initcond(t) => S113(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S113(t))%new-525
  crpind114: lemma (forall t: initcond(t) => S114(t))
                          and forall s, t: (indc2(s)
                          and trans(s, t) => S114(t))% bew-604 we need another:
  S150(t): bool = forall (i: index): next(t)(i) = r9
                implies rdcnt(t) = rd(t)
```

```
  indc3(t): bool =  indc2(t) and S150(t)
  crpind114i: lemma (forall t: initcond(t) => S114(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S114(t))% new 792
  crpind150: lemma (forall t: initcond(t) => S150(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S150(t))%  new-1477
  crpind115: lemma (forall t: initcond(t) => S115(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S115(t)) %new-786
  crpind121: lemma (forall t: initcond(t) => S121(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S121(t)) %new-916(gk)
  crpind122: lemma (forall t: initcond(t) => S122(t)) %no prf for those 3
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S122(t)) %need r5 => rdcnt>=1
  crpind123: lemma (forall t: initcond(t) => S123(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S123(t)) %r6=>rdcnt >= 1
  crpind124: lemma (forall t: initcond(t) => S124(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S124(t))%need waitAtPm2
                                                   % => rdcnt>=1
  crpind125: lemma (forall t: initcond(t) => S125(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S125(t)) %new-1296 (s "r")
  crpind131: lemma (forall t: initcond(s) => S131(s))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S131(t)) %new - 1063(s "r")
  crpind132: lemma (forall t: initcond(t) => S132(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S132(t))%new-1200 (s "r")
  crpind133: lemma (forall t: initcond(t) => S133(t))
                           and forall s, t: (indc3(s)
                           and trans(s, t) => S133(t))%new-1160(s "r")

  END invj1
```

7. cardsem.pvs

```
cardsem  : THEORY

  BEGIN
  importing invj1
  s: VAR stateneop
  t: VAR state
  i: VAR index
  P(t, i): bool = next(t)(i) = r3 or next(t)(i) = r4
                       or next(t)(i) = r5 or next(t)(i) = r6
         or next(t)(i) = r7 or next(t)(i) = waitAtPwr
                       or next(t)(i) = rlseAtPwr or next(t)(i) = waitAtPm2
         or  next(t)(i) = rlseAtPm2
  au(t): finite_set[index] = {i: index | P(t, i)}
  aux: lemma forall (t: state): (exists (i: index): next(t)(i) = r4
                           or next(t)(i) = r5 or next(t)(i) = r6
                           or next(t)(i) = r7 or next(t)(i) = waitAtPwr
                           or next(t)(i) = rlseAtPwr or next(t)(i) = waitAtPm2
                           or next(t)(i) = rlseAtPm2) => card(au(t)) >= 1 %new
  a(t): bool = card(au(t)) = rdcnt(t)
  indc4(t): bool = indc3(t) and a(t)
  a_inv: lemma (forall t: initcond(t) => a(t))
                           and forall s, t: (indc4(s)
                           and trans(s, t) => a(t)) %new-348
  crpind122i: lemma (forall t: initcond(t) => S122(t))
                           and forall s, t: (indc4(s)
                           and trans(s, t) => S122(t)) %new-130
```

```
  crpind123i: lemma (forall t: initcond(t) => S123(t))
                            and forall s, t: (indc4(s)
                            and trans(s, t) => S123(t)) %new-130
  crpind124i: lemma (forall t: initcond(t) => S124(t))
                            and forall s, t: (indc4(s)
                            and trans(s, t) => S124(t)) % new-140
  %the proofs of the three previous theorems pretty much alike
  END cardsem
```

8. dq.pvs

```
dq: THEORY

  BEGIN
  importing cardsem
  s: VAR stateneop
  s1, t, u: VAR state
  i: VAR index
  DQdecrease(s, t): bool = (exists i: IntRW(next(s)(i)) > IntRW(next(t)(i)))
                     and (forall i: IntRW(next(s)(i)) >= IntRW(next(t)(i)))
  dqa: theorem indc4(s) => (trans(s, t) and  not (m(s) = m(t) and
               w(s) = w(t) and rdcnt(s) = rdcnt(t) and
               (forall i: next(s)(i) = next(t)(i)) and
               rd(s) = rd(t) and wt(s) = wt(t)) => DQdecrease(s, t)) %new-1500
   dqb: lemma forall s1: (indc4(s1) =>
                       ((forall i: IntRW(next(s1)(i)) = 0) or
                       (exists t: (trans(s1, t) and
                       (not (m(s1) = m(t) and
                       w(s1) = w(t) and rdcnt(s1) = rdcnt(t) and
                       (forall i: next(s1)(i) = next(t)(i)) and
                       rd(s1) = rd(t) and wt(s1) = wt(t)) or
                       (exists u: (trans(t, u) and not (m(t) = m(u) and
                       w(t) = w(u) and rdcnt(t) = rdcnt(u) and
                       (forall i: next(t)(i) = next(u)(i)) and
                       rd(t) = rd(u) and wt(t) = wt(u))))))))
  %the previous unprovable, need dqbinv1 and Ssetm1,... as below
  dqc: theorem indc4(t) => (forall (i: index): (IntRW(next(t)(i)) = 0))
     implies (forall (i: index): (next(t)(i) /= waitAtPm1
     and  next(t)(i) /= waitAtPm2 and next(t)(i) /= waitAtPwr
     and  next(t)(i) /= waitAtPww))          %new-3s
  dqbinv1(t):bool = forall i: (next(t)(i) = waitAtPm1 or
                       next(t)(i) = waitAtPm2 or next(t)(i) = waitAtPwr or
                                    next(t)(i) = waitAtPww)=>
                 exists (j: index): (next(t)(j) /= waitAtPm1 and
                  next(t)(j) /= waitAtPm2 and next(t)(j) /= waitAtPwr and
                  next(t)(j) /= waitAtPww and next(t)(j) /= EOP)
  dqb2i: lemma (forall t: initcond(t) => dqbinv1(t))
                            and forall s, t: (dqbinv1(s) and indc4(s)
                            and trans(s, t) => dqbinv1(t)) %we need:
          %dqbinv2 and dqbinv3
 dqbinv2(t): bool =       cnt(w(t)) <= 0
                              => exists i:
                       (next(t)(i) = rlseAtPm1  and rd(t) >= 1) OR
                        (next(t)(i) = r2  and rd(t) >= 1) OR
                        (next(t)(i) = r3  and rd(t) >= 1) OR
                       next(t)(i) = rlseAtPwr or next(t)(i) = r4 or
                     next(t)(i) = r5 or next(t)(i) = r6 or
                     next(t)(i) = rlseAtPm2 or next(t)(i) = r7 or
    next(t)(i) = r8 or next(t)(i) = rlseAtPww or
                     (next(t)(i) = r9  and rd(t) >= 1 and cnt(m(t)) < 0)
                     or next(t)(i) = w2 or next(t)(i) = w3
 dqbinv3(t): bool =       cnt(m(t)) <= 0
                              => exists i:
                       next(t)(i) = rlseAtPm1 or next(t)(i) = r2 or
                     next(t)(i) = r3 or next(t)(i) = rlseAtPwr or
                     next(t)(i) = r4 or next(t)(i) = rlseAtPm2 or
```

```
   next(t)(i) = r7 or next(t)(i) = r8 or next(t)(i) = r9 or
                  (next(t)(i) = rlseAtPww and cnt(w(t)) < 0) or
                  (next(t)(i) = w2 and cnt(w(t)) < 0) or
                  (next(t)(i) = w3 and cnt(w(t)) < 0)
  dqb2ii: lemma (forall t: initcond(t) => dqbinv2(t))
                           and forall s, t: (dqbinv2(s) and indc4(s)
                           and trans(s, t) => dqbinv2(t))
  %Ssetm(t): bool = cnt(m(t)) < 0 => exists i: set(m(t))(i)
  Ssetm1(t): bool = forall i :set(m(t))(i) <=> next(t)(i) = waitAtPm1
                  or next(t)(i) = waitAtPm2
  %Ssetm2(t): bool = forall i: next(t)(i) = waitAtPm1
                  %or next(t)(i) = waitAtPm2 => set((m(t)))(i)
  %Ssetw(t): bool = cnt(w(t)) < 0 => exists i: set(w(t))(i)
  Ssetw1(t): bool = forall i :set(w(t))(i) <=> next(t)(i) = waitAtPww
                  or next(t)(i) = waitAtPwr
  %Ssetw2(t): bool = forall i: next(t)(i) = waitAtPww
                  %or next(t)(i) = waitAtPwr => set(w(t))(i)
  Ssetc(t): bool = cnt(m(t)) <= 0 => card(set(m(t))) = abs(cnt(m(t)))
  Ssetc1(t): bool = cnt(m(t)) =  1 => card(set(m(t))) = 0
  Ssetc2(t): bool = cnt(w(t)) <= 0 => card(set(w(t))) = abs(cnt(w(t)))
  Ssetc3(t): bool = cnt(w(t)) =  1 => card(set(w(t))) = 0
  indc5(t): bool = indc4(t) and dqbinv1(t) and dqbinv2(t) and dqbinv3(t)
                           and Ssetm1(t) and Ssetw1(t) and Ssetc(t)
                           and Ssetc1(t) and Ssetc2(t) and Ssetc3(t)
 END dq
```

9. dqb.pvs

```
dqb: THEORY

  BEGIN
  importing dq
  s: VAR stateneop
  t: VAR state
  i: VAR index
  %Ssetc(t): bool = cnt(m(t)) <= 0 => card(set(m(t))) = abs(cnt(m(t)))
  Ssetm1ind: lemma (forall t: initcond(t) => Ssetm1(t))
                           and forall s, t: (indc5(s) and
                           trans(s, t) => Ssetm1(t))  %new-1172<=>
  Ssetcind: lemma (forall t: initcond(t) => Ssetc(t))
                           and forall s, t: (indc5(s)
                           and Ssetc(s)
                           and trans(s, t) => Ssetc(t))
  %unprovable, we need another one for the previous:
  %Ssetc1(t): bool = cnt(m(t)) =  1 => card(set(m(t))) = 0
  Ssetcind1: lemma (forall t: initcond(t) => Ssetc(t))
                           and forall s, t: (indc5(s)
                           and trans(s, t) => Ssetc(t)) %new-67
  Ssetc1ind: lemma (forall t: initcond(t) => Ssetc1(t))
                           and forall s, t: (indc5(s)
                           and trans(s, t) => Ssetc1(t))%new-52
  %Ssetc2(t): bool = cnt(w(t)) <= 0 => card(set(w(t))) = abs(cnt(w(t)))
  Ssetw1ind: lemma (forall t: initcond(t) => Ssetw1(t))
                           and forall s, t: (indc5(s)
                           and trans(s, t) => Ssetw1(t))  %new-1155
  %unprovable, we need another one for the previous:
  %Ssetc3(t): bool = cnt(w(t)) =  1 => card(set(w(t))) = 0
  Ssetc2ind1: lemma (forall t: initcond(t) => Ssetc2(t))
                           and forall s, t: (indc5(s)
                           and trans(s, t) => Ssetc2(t)) %new 100
  Ssetc3ind: lemma (forall t: initcond(t) => Ssetc3(t))
                           and forall s, t: (indc5(s)
                           and trans(s, t) => Ssetc3(t))
  dqbinv1i: lemma (forall t: initcond(t) => dqbinv1(t))
                           and forall s, t: (indc5(s)
                           and trans(s, t) => dqbinv1(t)) %new 852
```

```
  dqbinv2: lemma (forall t: initcond(t) => dqbinv2(t))
                            and forall s, t: (indc5(s)
                            and trans(s, t) => dqbinv2(t))%unprovable, we need:
  P1(t, i): bool = next(t)(i) = r4
                       or next(t)(i) = r5 or next(t)(i) = r6
         or next(t)(i) = r7 or next(t)(i) = r8
                       or next(t)(i) = waitAtPm2
         or  next(t)(i) = rlseAtPm2
  cr(t): finite_set[index] = {i: index | P1(t, i)}
  craux: lemma forall (t: state): (exists (i: index): next(t)(i) = r4
                            or next(t)(i) = r5 or next(t)(i) = r6
                            or next(t)(i) = r7 or next(t)(i) = r8
                            or next(t)(i) = waitAtPm2
                            or next(t)(i) = rlseAtPm2) => card(cr(t)) >= 1
  cr1(t):bool = card(cr(t)) = rd(t)
  indc6(t): bool = indc5(t) and cr1(t)
  crinv: lemma (forall t: initcond(t) => cr1(t))
                            and forall s, t: (indc6(s)
                            and trans(s, t) => cr1(t)) %new 569
  dqbinv2final: lemma (forall t: initcond(t) => dqbinv2(t))
                            and forall s, t: (indc6(s)
                            and trans(s, t) => dqbinv2(t)) %new-1777
  dqbinv3: lemma (forall t: initcond(t) => dqbinv3(t))
                            and forall s, t: (indc6(s)
                            and trans(s, t) => dqbinv3(t)) %for the last goal
                                 %we need dqinv4
  dqinv4(t): bool = forall i: (next(t)(i) = rlseAtPww or next(t)(i) = w2
                    or next(t)(i) = w3) and cnt(w(t)) < 0 and cnt(m(t)) <= 0
                    and (forall (k: index): next(t)(k) /= waitAtPwr)
                   => exists (k: index):
                      next(t)(k) = rlseAtPm1 or next(t)(k) = r2 or
                      next(t)(k) = r3 or next(t)(k) = rlseAtPwr or
                      next(t)(k) = r4 or next(t)(k) = rlseAtPm2 or
                      next(t)(k) = r7 or next(t)(k) = r8 or
                      next(t)(k) = r9
  indc7(t): bool = indc6(t) and dqinv4(t)
  dqbinv3final: lemma (forall t: initcond(t) => dqbinv3(t))
                           and forall s, t: (indc7(s)
                            and trans(s, t) => dqbinv3(t)) %new - 756
  dqinv4: lemma (forall t: initcond(t) => dqinv4(t))
                            and forall s, t: (indc7(s)
                            and trans(s, t) => dqinv4(t)) %new-2594, with indc7
  END dqb
```

10. dqbfinal.pvs

```
dqbfinal  % [ parameters ]
: THEORY

  BEGIN

  % ASSUMING
   % assuming declarations
  % ENDASSUMING

  IMPORTING dqb
  s: VAR stateneop
  s1, t, u: VAR state
  i: VAR index
  %in order to prove TCC for dqb
  dqbinv5(t): bool = next(t)(pID(t)) /= EOP or forall i: next(t)(i) = EOP
  indc8(t): bool = indc7(t) and dqbinv5(t)
  dqbinv5final: lemma (forall t: initcond(t) => dqbinv5(t))
                            and forall s, t: (indc8(s)
                            and trans(s, t) => dqbinv5(t))
  dqbassist1: lemma nonempty?({i: index |
```

```
                        next(t)(i) /= waitAtPm1
                    AND next(t)(i) /= waitAtPm2
                    AND next(t)(i) /= waitAtPwr
                    AND next(t)(i) /= waitAtPww
                    AND next(t)(i) /= EOP}) and (LET rdcnt = rdcnt(t),
        pID =
          choose({i: index |
                        next(t)(i) /= waitAtPm1
                    AND next(t)(i) /= waitAtPm2
                    AND next(t)(i) /= waitAtPwr
                    AND next(t)(i) /= waitAtPww
                    AND next(t)(i) /= EOP}),
        rd = rd(t),
        wt = wt(t),
        cntm = cnt(m(t)),
        setm = set(m(t)),
        cntw = cnt(w(t)),
        setw = set(w(t)),
        next = next(t)
       IN
       (# pID := pID,
          m := (# cnt := cntm, set := setm #),
          w := (# cnt := cntw, set := setw #),
          rdcnt := rdcnt,
          next := next,
          rd := rd,
          wt := wt #))
     = u and indc8(t) => indc8(u)
dqbassist2: lemma forall s: indc8(s) => (p17(s)  =>
             (exists t: trans(s, t) and not (m(s) = m(t) and
             w(s) = w(t) and rdcnt(s) = rdcnt(t) and
             (forall i: next(s)(i) = next(t)(i)) and
             rd(s) = rd(t) and wt(s) = wt(t)))) %
dqbassist3: lemma forall s: indc8(s) => (p27(s)  =>
             (exists t: trans(s, t) and not (m(s) = m(t) and
             w(s) = w(t) and rdcnt(s) = rdcnt(t) and
             (forall i: next(s)(i) = next(t)(i)) and
             rd(s) = rd(t) and wt(s) = wt(t))))             %new-120
dqbassist4: lemma forall s: indc8(s) => (p32(s)  =>
               (exists t: trans(s, t) and not (m(s) = m(t) and
             w(s) = w(t) and rdcnt(s) = rdcnt(t) and
             (forall i: next(s)(i) = next(t)(i)) and
             rd(s) = rd(t) and wt(s) = wt(t))))                %120
dqbassist5: lemma forall s: indc8(s) => (p41(s)  =>
                  (exists t: trans(s, t) and not (m(s) = m(t) and
                  w(s) = w(t) and rdcnt(s) = rdcnt(t) and
                  (forall i: next(s)(i) = next(t)(i)) and
                  rd(s) = rd(t) and wt(s) = wt(t))))            %181
dqbassist6: lemma forall s: indc8(s) => (p31(s) or p40(s) =>
                    (exists t: trans(s, t) and not (m(s) = m(t) and
                      w(s) = w(t) and rdcnt(s) = rdcnt(t) and
                    (forall i: next(s)(i) = next(t)(i)) and
                    rd(s) = rd(t) and wt(s) = wt(t))))        %153
dqbassist: lemma forall s: indc8(s) =>
              (p4(s) or p13(s) or p22(s) or p36(s)
               => (exists t: (trans(s, t) and exists u: trans(t, u) and
                    not (m(t) = m(u) and
                    w(t) = w(u) and rdcnt(t) = rdcnt(u) and
                   (forall i: next(t)(i) = next(u)(i)) and
                   rd(t) = rd(u) and wt(t) = wt(u)))))
dqb: lemma forall s1: (indc8(s1) =>
                       ((forall i: IntRW(next(s1)(i)) = 0) or
                        (exists t: (trans(s1, t) and
        (not (m(s1) = m(t) and
            w(s1) = w(t) and rdcnt(s1) = rdcnt(t) and
            (forall i: next(s1)(i) = next(t)(i)) and
            rd(s1) = rd(t) and wt(s1) = wt(t)) or
        (exists u:
```

```
                (trans(t, u) and not (m(t) = m(u) and
            w(t) = w(u) and rdcnt(t) = rdcnt(u) and
            (forall i: next(t)(i) = next(u)(i)) and
            rd(t) = rd(u) and wt(t) = wt(u))))))))   %new-365justrerun
  END dqbfinal
```

11. ordering

```
ordering
: THEORY
  BEGIN
  importing dqbfinal
  s: VAR stateneop
  t: VAR  state
  active(l: label): nat = if l = EOP then 0
                          else 1
                          endif
  SUM(t: state, i: index): RECURSIVE nat =
    if i = 1 then  IntRW(next(t)(1))
    else IntRW(next(t)(i)) + SUM(t, i-1)
    endif
    measure i
  Pos(t: state, i: index): RECURSIVE nat =
    if i = 1 then active(next(t)(1))
    else active(next(t)(i)) + Pos(t, i-1)
    endif
    measure i
  DQtotal(s, t): bool= table
           %-----------------------------------%
           | Pos(s, M) > Pos(t, M) | TRUE             ||
           %-----------------------------------%
           | Pos(s, M) = Pos(t, M)
              and SUM(s, M) > SUM(t, M)|  TRUE        ||
           %-----------------------------------%
           | Pos(s, M) = Pos(t, M)
              and SUM(s, M) <= SUM(t, M)| FALSE      ||
           %-----------------------------------%
           | Pos(s, M) < Pos(t, M)       | FALSE      ||
           %-----------------------------------%
  endtable
  partot: lemma DQdecrease(s, t) => DQtotal(s, t)
  END ordering
```

12. pvs-strategies

```
%finding invariants
(defstep get_inv ()
(branch (split)
(
(then
(skolem!)
(flatten)
(typepred "next(t!1)")
(inst - "pID(t!1)")
(flatten)
        (ind_inv1$)
(branch (split +)
((then
                (ind_inv1$)
(try (skolem!)
(then
(expand "initcond")
(flatten)
(inst - "i!1")
(grind))(grind))))))
```

```
(then
(skolem!)
(flatten)
(ind_inv1$)
                (typepred "next(s!1)")
(inst - "pID(s!1)")
(flatten)
                (expand* "ind7" "ind6" "ind5" "ind4"
                         "ind3" "ind2" "ind1" "ind" )
                (branch (split +)
        ((then
(ind_inv1$)
(try (skolem!)
(then
(flatten)
(inst - "i!1")
(branch (case "i!1=pID(s!1)")
((then
                (expand "trans")
                (branch (tasimp)
((if (equal (get-goalnum *ps*) 30)
                    (then (lemma "trans_TCC2")
                     (branch (inst - "pID(s!1)" "s!1" "s!1")
                     ((branch (split -1)((if (equal (get-goalnum *ps*) 30)
                     (then (grind))(postpone))))(then (reveal -2)
                     (hide  -3 -4 -5 -6 -7 -8 -9 -10 +)(grind)))))
                     (then (inst - "pID(s!1)")(grind))))))(skip))))
                                          (grind))))))))
"" "")
(defstep ind_inv1 ()
(let ((sforms (s-forms (current-goal *ps*)))
 (inv_name (string  (id (operator
                     (formula (car (select-seq sforms 1))))))))
     (expand inv_name))
"" "")
(defstep ind_inv2 ()
(let ((sforms (s-forms (current-goal *ps*)))
 (inv_name (string  (id (operator
                     (formula (car (select-seq sforms 2))))))))
     (expand inv_name))
"" "")
(defstep bddtrans ()
  (let
    ((transvar
       (gather-fnums
         (s-forms *goal*)
         '-
         nil
         #'(lambda (sf)
             (and (negation? (formula sf)) (branch? (args1 (formula sf)))))))
     )
       (bddsimp transvar)
  )
"..."
"..."
)
%invariants of type forall (i:index): P(x) => v
(defstep s (arg)
 (branch
 (split)
((then (skolem!)
  (flatten)
  (expand "initcond")
  (flatten)
  (ind_inv1$)
  (skolem!)
  (inst - "i!1")
  (grind)
```

```
        )
        (then
(skolem!)
(flatten)
        (let ((sforms (s-forms (current-goal *ps*)))
 (indinv (string  (id (operator (args1
                    (formula (car (select-seq sforms -1)))))))))
        (then (if (equal indinv "indc3") (expand* "indc3" "indc2"
               "indc1" "indc") (if (equal indinv "indc2")
               (expand* "indc2" "indc1" "indc")(if (equal indinv "indc1")
                (expand* "indc1" "indc")(expand "indc"))))
(flatten)
(ind_inv1$)
(skolem!)
(inst?)
(flatten)
(branch
(case "i!1=pID(s!1)")
((then
                (hide -2 -3)
(expand "trans")
                (branch (tasimp)
((if (equal (get-goalnum *ps*) 30)
                    (then (lemma "trans_TCC2")
                          (branch (inst - "pID(s!1)" "s!1" "s!1")
                          ((branch (split -1)((if (equal (get-goalnum *ps*) 30)
          (then (grind))(postpone))))(then (reveal -4)(hide-all-but
                                :keep-fnums
                                (-1 -2 -12 -13 -14 -15 -16 -24
                                -26 -35 -27 -28))(grind)))))(then
                                (inst - "pID(s!1)")(grind))))))
(then
        (expand "trans")
        (if (equal arg "r")(then (hide -2)(expand* "CS1" "CS1pred")
                                         (inst - "i!1" "pID(s!1)"))
                           (then (hide -1)(expand* "CS2" "CS2pred")
                                         (inst - "i!1" "pID(s!1)")))
        (branch (tasimp)
((if (equal (get-goalnum *ps*) 30)
                    (then (lemma "trans_TCC2")
                          (branch (inst - "pID(s!1)" "s!1" "s!1")
                          ((branch (split -1)((if (equal (get-goalnum *ps*) 30)
          (then (inst - "i!1")(grind))(propax))))(then (reveal -4)(hide-all-but
                                :keep-fnums
                                (-1 -2 -12 -13 -14 -15 -16 -24
                                 -26 -27 -28 -35))(grind)))))(then
                                (inst - "i!1")(grind))))))))))))
"" "")
(defstep s1 ()
 (branch
 (split)
((grind)
        (then
(skolem!)
(flatten)
        (typepred "next(s!1)")
        (inst - "pID(s!1)")
        (let ((sforms (s-forms (current-goal *ps*)))
 (indinv (string  (id (operator (args1
                    (formula (car (select-seq sforms -2)))))))))
        (then (if (equal indinv "indc3") (then (expand "indc3")(expand "indc2")
               (expand "indc1")(expand "indc")) (if (equal indinv "indc2")
               (then (expand "indc2")
               (expand "indc1")(expand "indc"))(if (equal indinv "indc1")
                (then (expand "indc1")(expand "indc"))(expand "indc"))))
        (flatten)
        (grind))))))
"" "")
```

```
(defstep ind_invs ()
(let ((sforms (s-forms (current-goal *ps*)))
 (inv_name (string  (id (operator
                   (formula (car (select-seq sforms 2))))))))
     (expand inv_name))
"" "")
(defstep ref_induct()
(let ((sforms (s-forms (current-goal *ps*)))
 (refStepName (string (id (formula (car sforms))))))
 (then (expand refStepName)
         (split)))
"" "")
(defstep tasimp ()
  (let
    ((transvar (car
       (gather-fnums
         (s-forms *goal*)
         '-
         nil
         #'(lambda (sf)
             (and (negation? (formula sf)) (branch? (args1 (formula sf))))))))
     )
        (then (branch (split transvar)((then (flatten)(skip))(repeat*
              (if (equal (get-goalnum *ps*) 1)
               (then (flatten)(skip))(then (flatten)
               (branch (split -1)((skip)(skip)))))))))
"..."
"..."
)
(defstep s_tcc ()
(then
             (flatten)
             (hide -1)
             (skolem!)
             (flatten)
             (let ((sforms (s-forms (current-goal *ps*)))
        (indinv (string  (id (operator (args1
                   (formula (car (select-seq sforms -1))))))))
             (then (expand indinv)(if (equal indinv "indc8")
                                 (expand* "indc8" "indc7" "indc6" "indc5"
                                  "indc4" "indc3" "indc2" "indc1" "indc")
                               (if (equal indinv "indc7")
                                    (expand* "indc7" "indc6" "indc5"
                                    "indc4" "indc3" "indc2" "indc1" "indc")
                                (if (equal indinv "indc6")
                                       (expand* "indc6" "indc5"
                                       "indc4" "indc3" "indc2" "indc1" "indc")
                                  (if (equal indinv "indc5")
                                         (expand*  "indc5"
                                         "indc4" "indc3" "indc2" "indc1" "indc")
                                  (if (equal indinv "indc4")
                                           (expand*
                                         "indc4" "indc3" "indc2" "indc1" "indc")
                                 (if (equal indinv "indc3")
                                           (expand* "indc3" "indc2" "indc1" "indc")
                                   (if (equal indinv "indc2")
                                       (expand* "indc2" "indc1" "indc")
                                    (if (equal indinv "indc1")
               (expand* "indc1" "indc")(expand "indc"))))))))))
             (flatten)
             (s_tcc_aux$)
             (expand* "p1" "p7" "p10" "p12" "p15"
                      "p19" "p25" "p30" "p33" "p39")
             (grind))))
"" "")
(defstep s_tcc_aux ()
  (let
    ((transvar (car
```

```
       (gather-fnums
         (s-forms *goal*)
         '-
         nil
         #'(lambda (sf)
             (and (negation? (formula sf)) (disjunction?
                          (args1 (formula sf))))))))
     )
        (hide-all-but :keep-fnums (-3 -4 -5 -6 -7 -15 -17 -18 -19 -26
                                   transvar)))
"..."
"..."
)
%semaphore invariants
(defstep css (inv)
 (let ((invname (concatenate 'string inv "pred")))
 (branch
 (split)
((then (skolem!)
  (flatten)
  (expand inv)
  (expand "initcond")
  (flatten)
  (skolem!)
  (inst - "i!1")
  (grind)
       )
       (then
(skolem!)
(flatten)
(expand* "indc1" "indc")
(flatten)
(expand* inv invname)
(ind_inv1$)
(skolem!)
(inst-cp - "i!1" "j!1")
(flatten)
       (expand* "S31" "S32" "S33" "S34" "S35" "S36" "S37" "S38" "S39"
                "S41" "S42" "S81" "S82" "S83" "S10" "S101" "S111" "S112"
                "S113" "S114" "S115" "S121" "S122" "S123" "S124" "S125"
                "S131"  "S132" "S133" "S91" "S92" "S140")
(branch
(case "i!1=pID(s!1)")
((then
(if (equal inv "CS1") (hide -4 -6) (hide -2 -3))
                (repeat* (inst - "j!1"))
(expand "trans")
                (branch (tasimp)
                ((if (equal (get-goalnum *ps*) 30)
                    (then (lemma "trans_TCC2")
                          (branch (inst - "pID(s!1)" "s!1" "s!1")
                          ((branch (split -1)((if (equal (get-goalnum *ps*) 30)
         (then (inst-cp - "pID(s!1)")(inst - "j!1")(grind))(propax))))
                          (then (hide-all-but :keep-fnums  -1)
                           (if (equal inv "CS1")
                             (reveal -12 -13 -25 -26 -27 -66 -74 -82 -83 -85)
                                 (reveal -11 -12 -13 -25 -26 -27 -66 -74 -82 -83))
                                    (grind)))))
                            (then (inst-cp - "pID(s!1)")(inst - "j!1")(grind))))))
(then
         (if (equal inv "CS1") (hide -5) (hide -1))
        (inst-cp - "i!1" "pID(s!1)")
(inst - "j!1" "pID(s!1)")
(repeat* (inst - "i!1"))
        (expand "trans")
        (branch (tasimp)
        ((if (equal (get-goalnum *ps*) 30)
                    (then (lemma "trans_TCC2")
```

```
                          (branch (inst - "pID(s!1)" "s!1" "s!1")
                          ((branch (split -1)((if (equal (get-goalnum *ps*) 30)
            (then (inst-cp - "j!1")(inst - "i!1")(grind))(propax))))
               (then (hide-all-but :keep-fnums -1)
                 (if (equal inv "CS1")
                      (reveal -13 -14 -26 -27 -28 -67 -75 -83 -84 -86)
                       (reveal -12 -13 -14 -26 -27 -28 -67 -75 -83 -84))
                           (grind)))))
                (then (inst-cp - "j!1")(inst - "i!1")(grind)))))))))))
    "" "")
```

C.4 The List of All Auxiliary Invariants

```
t: VAR state
i, j: VAR index
S7(t): bool = (rdcnt(t) >= 0)
S2(t): bool = (cnt(w(t)) <=  1)
S1(t): bool = (cnt(m(t)) <=  1)
S6(t): bool = (cnt(w(t)) = 1 => (wt(t) = 0 and rd(t) = 0))
S91(t): bool = forall (i: index): next(t)(i) = rlseAtPwr
        implies wt(t) = 0
S92(t): bool = forall (i: index): next(t)(i) = rlseAtPwr
        implies cnt(w(t)) <= 0
S33(t): bool = forall (i: index):
       (next(t)(i) = rlseAtPm1)
          implies cnt(m(t)) <= 0
S34(t): bool = forall (i: index):
       (next(t)(i) = r2) implies cnt(m(t)) <= 0
S35(t): bool = forall (i: index):
       (next(t)(i) = r3)
          implies cnt(m(t)) <= 0
S31(t): bool = forall (i: index):
       next(t)(i) = r4
          implies cnt(m(t)) <= 0
S36(t): bool = forall (i: index):
       (next(t)(i) = rlseAtPm2)
          implies cnt(m(t)) <= 0
S37(t): bool = forall (i: index):
       next(t)(i) = r7
          implies cnt(m(t)) <= 0
S38(t): bool = forall (i: index):
       next(t)(i) = r8
          implies cnt(m(t)) <= 0
S32(t): bool = forall (i: index):
       next(t)(i) = r9
          implies cnt(m(t)) <= 0
S39(t): bool = forall (i: index):
       next(t)(i) = rlseAtPwr
          implies cnt(m(t)) <= 0
S41(t): bool = forall (i: index): next(t)(i) = r3
        implies rdcnt(t) = rd(t) +  1
S42(t): bool = forall (i: index):  next(t)(i) = rlseAtPwr
        implies rdcnt(t) = rd(t) +  1
S5(t): bool = forall (i: index):  next(t)(i) = r8
              implies rd(t) = rdcnt(t) + 1
S81(t): bool = forall (i: index): next(t)(i) = rlseAtPww
               implies wt(t) = 0
S82(t): bool = forall (i: index): next(t)(i) = rlseAtPww
               implies rd(t) = 0
S83(t): bool = forall (i: index): next(t)(i) = rlseAtPww
               implies cnt(w(t)) < 1
S10(t): bool = forall (i: index): next(t)(i) = w2
             implies wt(t) = 1
```

```
S101(t): bool = forall (i: index): next(t)(i) = w3
             implies wt(t) = 1
S111(t): bool = forall (i: index): next(t)(i) = r7
              implies rd(t) = rdcnt(t)
S112(t): bool = forall (i: index): next(t)(i) = r2
              implies rd(t) = rdcnt(t)
S113(t): bool = forall (i: index): next(t)(i) = r4
              implies rd(t) = rdcnt(t)
S114(t): bool = forall (i: index): next(t)(i) = rlseAtPm2
              implies rd(t) = rdcnt(t)
S115(t): bool = forall (i: index): next(t)(i) = rlseAtPm1
              implies rd(t) = rdcnt(t)
S121(t): bool = forall (i: index): next(t)(i) = r4
              implies rd(t) >= 1
S122(t): bool = forall (i: index): next(t)(i) = r5
              implies rd(t) >= 1
S123(t): bool = forall (i: index): next(t)(i) = r6
              implies rd(t) >= 1
S124(t): bool = forall (i: index): next(t)(i) = rlseAtPm2
              implies rd(t) >= 1
S125(t): bool = forall (i: index): next(t)(i) = r7
              implies rd(t) >= 1
S131(t): bool = forall (i: index): next(t)(i) = r1
             and cnt(m(t)) = 1
             implies rdcnt(t) = rd(t)
S132(t): bool = forall (i: index): next(t)(i) = r5
              and cnt(m(t)) = 1
             implies rdcnt(t) = rd(t)
S133(t): bool = forall (i: index): next(t)(i) = r6
              and cnt(m(t)) = 1
             implies rdcnt(t) = rd(t)
S140(t): bool = forall i: next(t)(i) = waitAtPwr
             implies cnt(m(t)) <= 0
S43(t): bool = forall i: next(t)(i) = waitAtPwr
             implies rdcnt(t) = rd(t) + 1
S150(t): bool = forall i: next(t)(i) = r9
             implies rdcnt(t) = rd(t)
a(t):bool = card(au(t)) = rdcnt(t), where
            au(t): finite_set[index] = {i: index | P(t, i)}
            P(t, i): bool = next(t)(i) = r3 or next(t)(i) = r4
                     or next(t)(i) = r5 or next(t)(i) = r6
      or next(t)(i) = r7 or next(t)(i) = waitAtPwr
                     or next(t)(i) = rlseAtPwr or next(t)(i) = waitAtPm2
      or  next(t)(i) = rlseAtPm2
CS1(t): bool =
            (forall (i, j: index): CS1pred(t, i) and
               CS1pred(t, j) => i = j), where
                   CS1pred(t, i): bool =  next(t)(i) = rlseAtPm1
                        or next(t)(i) = r2 or
                        next(t)(i) = r3 or next(t)(i) = r4
                        or next(t)(i) = rlseAtPm2
                        or next(t)(i) = r7 or next(t)(i) = r8
                        or next(t)(i) = r9 or next(t)(i) = waitAtPwr
                        or next(t)(i) = rlseAtPwr
CS2(t): bool =
            (forall (i, j: index): CS2pred(t, i) and CS2pred(t, j)
            => i = j), where
                       CS2pred(t, i): bool =  next(t)(i) = w2
                                     or next(t)(i) = w3 or
                    next(t)(i) = rlseAtPwr or next(t)(i) = rlseAtPww
%The additional invariants needed for the clean completion proof:
Ssetm1(t): bool = forall i :set(m(t))(i) <=> next(t)(i) = waitAtPm1
                or next(t)(i) = waitAtPm2
Ssetw1(t): bool = forall i :set(w(t))(i) <=> next(t)(i) = waitAtPww
                or next(t)(i) = waitAtPwr
Ssetc(t): bool = cnt(m(t)) <= 0 => card(set(m(t))) = abs(cnt(m(t)))
Ssetc1(t): bool = cnt(m(t)) =  1 => card(set(m(t))) = 0
Ssetc2(t): bool = cnt(w(t)) <= 0 => card(set(w(t))) = abs(cnt(w(t)))
```

```
Ssetc3(t): bool = cnt(w(t)) =  1 => card(set(w(t))) = 0
cr1(t):bool = card(cr(t)) = rd(t), where
         P1(t, i): bool = next(t)(i) = r4
                     or next(t)(i) = r5 or next(t)(i) = r6
      or next(t)(i) = r7 or next(t)(i) = r8
                     or next(t)(i) = waitAtPm2
      or  next(t)(i) = rlseAtPm2,
         cr(t): finite_set[index] = {i: index | P1(t, i)}
dqbinv1(t):bool = forall (i: index): (next(t)(i) = waitAtPm1 or
                     next(t)(i) = waitAtPm2 or next(t)(i) = waitAtPwr or
                                  next(t)(i) = waitAtPww)=>
              exists (j: index): (next(t)(j) /= waitAtPm1 and
               next(t)(j) /= waitAtPm2 and next(t)(j) /= waitAtPwr and
               next(t)(j) /= waitAtPww and next(t)(j) /= EOP)
dqbinv2(t): bool =       cnt(w(t)) <= 0
                           => exists i:
                     (next(t)(i) = rlseAtPm1  and rd(t) >= 1) OR
                      (next(t)(i) = r2  and rd(t) >= 1) OR
                      (next(t)(i) = r3  and rd(t) >= 1) OR
                     next(t)(i) = rlseAtPwr or next(t)(i) = r4 or
                  next(t)(i) = r5 or next(t)(i) = r6 or
                  next(t)(i) = rlseAtPm2 or next(t)(i) = r7 or
                  next(t)(i) = r8 or next(t)(i) = rlseAtPww or
                  (next(t)(i) = r9  and rd(t) >= 1 and cnt(m(t)) < 0)
                  or next(t)(i) = w2 or next(t)(i) = w3
dqbinv3(t): bool =       cnt(m(t)) <= 0
                           => exists i:
                     next(t)(i) = rlseAtPm1 or next(t)(i) = r2 or
                  next(t)(i) = r3 or next(t)(i) = rlseAtPwr or
                  next(t)(i) = r4 or next(t)(i) = rlseAtPm2 or
                  next(t)(i) = r7 or next(t)(i) = r8 or next(t)(i) = r9 or
                  (next(t)(i) = rlseAtPww and cnt(w(t)) < 0) or
                  (next(t)(i) = w2 and cnt(w(t)) < 0) or
                  (next(t)(i) = w3 and cnt(w(t)) < 0)
dqinv4(t): bool = forall i: (next(t)(i) = rlseAtPww or next(t)(i) = w2
                  or next(t)(i) = w3) and cnt(w(t)) < 0 and cnt(m(t)) <= 0
                  and (forall (k: index): next(t)(k) /= waitAtPwr)
                 => exists (k: index):
                    next(t)(k) = rlseAtPm1 or next(t)(k) = r2 or
                    next(t)(k) = r3 or next(t)(k) = rlseAtPwr or
                    next(t)(k) = r4 or next(t)(k) = rlseAtPm2 or
                    next(t)(k) = r7 or next(t)(k) = r8 or
                    next(t)(k) = r9
dqbinv5(t): bool = next(t)(pID(t)) /= EOP or forall i: next(t)(i) = EOP
```

C.5 Invariants From the Manual Proof of Readers/Writers Problem

```
rp1(t): bool = wt(t) = 0 or rd(t) = 0
rp2(t): bool = wt(t) < 2
V1(t): bool = rd(t) >= 0
V2(t): bool = wt(t) >= 0
V3(t): bool = (rdcnt(t) >= 0)
V4(t): bool = (cnt(w(t)) <=  1)
V5(t): bool = (cnt(m(t)) <=  1)
V6(t): bool = (cnt(w(t)) = 1 => (wt(t) = 0 and rd(t) = 0))
V7(t): bool = (rdcnt(t) > 1 => rd(t) >= 1)
V8(t): bool = (cnt(w(t)) < 1 => ((wt(t) = 1 and rd(t) = 0) or
                                 (rd(t) >= 1 and wt(t) = 0) or
                                 (rd(t) = 0 and wt(t) = 0 and
                  exists (i: index): (next(t)(i) = rlseAtPwr
                                 or next(t)(i) = rlseAtPww))))
```

```
V9(t): bool = (exists (i: index): (i = pID(t) and (next(t)(i) = r3 or
                                   next(t)(i) = rlseAtPwr or
                                   next(t)(i) = r4 or next(t)(i) = r5 or
                     next(t)(i) = r6 or next(t)(i) = rlseAtPm2
                                   or next(t)(i) = r7))) => rdcnt(t) > 0
V10(t): bool = ((exists (i: index): i = pID(t) and next(t)(i) = rlseAtPwr)
                                    implies (rd(t) = 0 and cnt(w(t)) < 1))
V11(t): bool = (exists (i: index): i = pID(t) and (next(t)(i) = rlseAtPm1 or
                                   next(t)(i) = r2 or next(t)(i) = r3 or
                                   next(t)(i) = rlseAtPwr or
                  next(t)(i) = r4 or
                                   next(t)(i) = rlseAtPm2 or
                                   next(t)(i) = r7 or next(t)(i) = r8 or
                                   next(t)(i) = r9)) implies cnt(m(t)) < 1
V12(t): bool = (exists (i: index): i = pID(t) and (next(t)(i) = r1 or
                                   next(t)(i) = rlseAtPm1 or
                                   next(t)(i) = r2 or next(t)(i) = r4 or
                  next(t)(i) = r5 or next(t)(i) = r6 or
                                   next(t)(i) = rlseAtPm2 or next(t)(i) = r7
                                   or next(t)(i) = r9)) implies
                                   rd(t) = rdcnt(t)
V13(t): bool = (exists (i: index): i = pID(t) and (next(t)(i) = r3 or
                                   next(t)(i) = rlseAtPwr ))  implies
                                   rd(t) = rdcnt(t) - 1
V14(t): bool = (exists (i: index): i = pID(t) and next(t)(i) = r8)  implies
                                   rd(t) = rdcnt(t) + 1
V15(t): bool = (exists (i: index): i = pID(t) and next(t)(i) = rlseAtPww)
                                   implies (wt(t) = 0  and cnt(w(t)) < 1)
V16(t): bool = (exists (i: index): i = pID(t) and (next(t)(i) = w3 or
                                   next(t)(i) = w2))  implies
                                   (wt(t) = 1 and cnt(w(t)) < 1)
```