INSPECTION OF CONCURRENT SYSTEMS
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

© Copyright by Vera Pantelic, December 22, 2005
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

Abstract i

Acknowledgements iv

List of Figures viii

List of Tables ix

1 Introduction 1
   1.1 Motivation 1
   1.2 Our Approach 1
   1.3 Contribution of the Thesis 3
   1.4 Structure of The Thesis 4

2 Inspection of Concurrent Programs 5
   2.1 Formal Modeling of Concurrency 5
   2.2 About Inspection Based on Tables 6
   2.3 The Inspection of Concurrent Programs Using Tables 6
      2.3.1 Introduction to the Approach 7
      2.3.2 Example Application: Readers/Writers Problem 8

3 Introduction to SPIN, SAL, and PVS 13
   3.1 The SPIN Model Checker 13
   3.2 SAL 14
   3.3 PVS 15
      3.3.1 The PVS Language and Proof Checker 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>The Sequent Calculus of PVS</td>
<td>16</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Tabular Specification of Functions</td>
<td>17</td>
</tr>
<tr>
<td>3.3.4</td>
<td>The PVS COND Construct</td>
<td>18</td>
</tr>
<tr>
<td>3.3.5</td>
<td>The PVS TABLE Construct</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Model Checking The Readers/Writers Problem</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Model Checking The Original Version In SPIN</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Specification in SPIN</td>
<td>22</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Analysis in SPIN</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Formalization of Readers/Writers Problem in SAL</td>
<td>25</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Specification in SAL</td>
<td>25</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Analysis in SAL</td>
<td>28</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Summary</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Theorem Proving in PVS</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>The Theory Hierarchy</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>The decl Theory</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>The table Theory</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>Verifying the Hand-Written Proof</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Verification in PVS Revisited</td>
<td>43</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Proof of the Safety Property</td>
<td>44</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Proof of the Theorem of Decreasing Quantity</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary</td>
<td>51</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>52</td>
</tr>
<tr>
<td>6.2</td>
<td>Limitations and Future Work</td>
<td>54</td>
</tr>
<tr>
<td>A</td>
<td>Specification of P/V Semaphore Operations</td>
<td>59</td>
</tr>
<tr>
<td>B</td>
<td>The Tabular Representation of the Rewritten Readers/Writers Program</td>
<td>61</td>
</tr>
<tr>
<td>C</td>
<td>The Readers/Writers Model in SPIN, SAL, and PVS</td>
<td>66</td>
</tr>
<tr>
<td>C.1</td>
<td>The Readers/Writers Model in SPIN</td>
<td>66</td>
</tr>
<tr>
<td>C.2</td>
<td>Model of Readers/Writers Program in SAL</td>
<td>67</td>
</tr>
</tbody>
</table>
CONTENTS

C.3 PVS files .................................................. 72
C.4 The List of All Auxiliary Invariants .................................. 96
C.5 Invariants From the Manual Proof of Readers/Writers Problem .... 98
List of Figures

2.1 Readers/Writers program rewritten .................................. 10
3.1 Sequents in sequent calculus ........................................... 17
3.2 COND construct and PVS interpretation .............................. 19
3.3 One-dimensional vertical table in PVS ................................. 20
4.1 Semaphore in SPIN ....................................................... 22
4.2 Modeling writer processes in SPIN ................................... 23
4.3 The context rw .......................................................... 26
4.4 Nondeterminism inside of the process module ....................... 27
4.5 The module main ........................................................ 28
5.1 The theory hierarchy ..................................................... 35
5.2 Theory decl ............................................................. 36
5.3 PVS definition of the function IntRW ................................. 37
5.4 Tabular representation of Readers/Writers problem in PVS ...... 38
A.1 Specification of P(sem) operation .................................... 59
A.2 Specification of V(sem) operation ..................................... 60
B.1 The tabular representation of the rewritten Readers/Writers Program 62
B.2 Figure B.1 continued ................................................... 63
List of Tables

2.1 The $IntRW$ function definition .................................. 11
2.2 The order property of DQ ........................................... 12
4.1 SPIN model checking results ........................................ 24
4.2 SAL model checking results ......................................... 32
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
1. Introduction

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. Inspection of Concurrent Programs

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:

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

WRITER: P(w);
    WRITE;
    V(w);

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:

$$(rd = 0 \lor wt = 0) \land wt < 2$$ (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]:

\[
DQ = (Pros, \text{IntRW}(\text{next}[1]), \text{IntRW}(\text{next}[2]), \ldots, \text{IntRW}(\text{next}[M]))
\]

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

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \text{IntRW}(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1 )</td>
<td>15</td>
</tr>
<tr>
<td>( \text{waitAtPm1} )</td>
<td>14</td>
</tr>
<tr>
<td>( \text{rlseAtPm1} )</td>
<td>13</td>
</tr>
<tr>
<td>( r2 )</td>
<td>12</td>
</tr>
<tr>
<td>( r3 )</td>
<td>11</td>
</tr>
<tr>
<td>( \text{waitAtPwr} )</td>
<td>10</td>
</tr>
<tr>
<td>( \text{rlseAtPwr} )</td>
<td>9</td>
</tr>
<tr>
<td>( r4 )</td>
<td>8</td>
</tr>
<tr>
<td>( r5 )</td>
<td>7</td>
</tr>
<tr>
<td>( r6 )</td>
<td>6</td>
</tr>
<tr>
<td>( \text{waitAtPm2} )</td>
<td>5</td>
</tr>
<tr>
<td>( \text{rlseAtPm2} )</td>
<td>4</td>
</tr>
<tr>
<td>( r7 )</td>
<td>3</td>
</tr>
<tr>
<td>( r8 )</td>
<td>2</td>
</tr>
<tr>
<td>( r9 )</td>
<td>1</td>
</tr>
<tr>
<td>( w1 )</td>
<td>5</td>
</tr>
<tr>
<td>( \text{waitAtPww} )</td>
<td>4</td>
</tr>
<tr>
<td>( \text{rlseAtPww} )</td>
<td>3</td>
</tr>
<tr>
<td>( w2 )</td>
<td>2</td>
</tr>
<tr>
<td>( w3 )</td>
<td>1</td>
</tr>
<tr>
<td>( \text{EOP} )</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: The \( \text{IntRW} \) function definition

Let \( l = 1, 2 \). Suppose that, at the state \( l \), the \( \text{next}_l \) is the value of \( \text{next} \), 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

\[
\sum r_i = \begin{cases} 
0, & n = 0 \\
\sum_{i=1}^{n} \text{IntRw}(\text{next}_i[i]), & 0 < n \leq M
\end{cases} \tag{2.2}
\]

\[
\sum w_i = \begin{cases} 
0, & n = M \\
\sum_{i=n+1}^{M} \text{IntRw}(\text{next}_i[i]), & 0 < n < M
\end{cases} \tag{2.3}
\]

\[DQ_i = (\text{Pros}_i, \text{IntRW}(\text{next}_i[i]), \ldots, \text{IntRW}(\text{next}_i[i])) \tag{2.4}\]

Then, the order property of DQ is given by the Table 2.2 where \( DQ\text{order} \) stands for \( DQ_1 > DQ_2 \).

<table>
<thead>
<tr>
<th>( DQ\text{order} )</th>
<th>( \text{Pros}_1 &gt; \text{Pros}_2 )</th>
<th>( \text{Pros}_1 = \text{Pros}_2 )</th>
<th>( \text{Pros}_1 &lt; \text{Pros}_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

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, $\neg P_1$, $P_1 \land Q_1$, $P_1 \lor Q_1$ 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, $P_1, P_2, \ldots, P_n \vdash Q_1 \lor Q_2 \lor \ldots \lor Q_m$, or equivalently, $P_1 \land P_2 \land \ldots \land P_n \vdash Q_1 \lor Q_2 \lor \ldots \lor Q_m$. 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.

\[
\begin{array}{c|c}
P_1, P_2, \ldots, P_n & P_1 \\
P_2 & P_2 \\
\vdots & \vdots \\
P_n & P_n \\
Q_1 \lor Q_2 \lor \ldots \lor Q_m & Q_1 \\
\vdots & \vdots \\
Q_m & Q_m
\end{array}
\]

Figure 3.1: Sequents in sequent calculus

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

\[
\begin{array}{c|c|c}
\vdots & \vdots & \vdots \\
P & \top & \bot \\
P & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\end{array}
\]

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

### 3.3.3 Tabular Specification of Functions

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

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

or

\[
\begin{bmatrix}
c_1 & e_1 \\
c_2 & e_2 \\
\vdots & \vdots \\
c_n & e_n
\end{bmatrix}
\]
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.,
   \[ i \neq j \Rightarrow \neg(c_i \land c_j). \]

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

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

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

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

which can be specified as a table:

<table>
<thead>
<tr>
<th>$x &lt; 0$</th>
<th>$x = 0$</th>
<th>$x &gt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### 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.
The following is the PVS definition of \(\text{sign}(x)\) function using the PVS COND construct:

\[
\text{signs: \text{TYPE} = \{ i: \text{int} \mid i \geq -1 \& i \leq 1 \}}
\]

\[
\text{sign\_cond}(x: \text{real}): \text{signs} =
\]

\[
\begin{array}{l}
\text{COND} \\
x < 0 \rightarrow -1, \\
x = 0 \rightarrow 0, \\
x > 0 \rightarrow 1
\end{array}
\]

\[
\text{ENDCOND}
\]

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

\[
\%
\text{Disjointness TCC generated (at line 11, column 1) for}
\%
\text{COND} \ x < 0 \rightarrow -1, \ x = 0 \rightarrow 0, \ x > 0 \rightarrow 1 \ \text{ENDCOND}
\]

\[
\text{sign\_cond\_TCC1: OBLIGATION}
\]

\[
\text{FORALL (x: real):
}\]

\[
\text{NOT (x < 0 AND x = 0) AND}
\]

\[
\text{NOT (x < 0 AND x > 0) AND NOT (x = 0 AND x > 0);
}\]

\[
\%
\text{Coverage TCC generated (at line 11, column 1) for}
\%
\text{COND} \ x < 0 \rightarrow -1, \ x = 0 \rightarrow 0, \ x > 0 \rightarrow 1 \ \text{ENDCOND}
\]

\[
\text{sign\_cond\_TCC2: OBLIGATION FORALL (x: real): x < 0 OR x = 0 OR x > 0;}
\]
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.

```pvs
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

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

\[(rd = 0 \lor wt = 0) \land wt < 2 \land rd \geq 0 \land wt \geq 0\]  \hspace{1cm} (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 \lor wt == 0) \&\& wt < 2 \&\& rd >= 0 \&\& wt >= 0) -> break
::: else
  od
}
```
4. Model Checking The Readers/Writers Problem

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.

<table>
<thead>
<tr>
<th></th>
<th>safety/completion states</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3R/2W</td>
<td>3619</td>
<td>0.02</td>
</tr>
<tr>
<td>5R/5W</td>
<td>0.4·10^6</td>
<td>1.25</td>
</tr>
<tr>
<td>6R/6W</td>
<td>2.3·10^6</td>
<td>115</td>
</tr>
<tr>
<td>8R/8W</td>
<td>8.4·10^7</td>
<td>6555</td>
</tr>
<tr>
<td>10R/10W</td>
<td>-</td>
<td>&gt;20h</td>
</tr>
</tbody>
</table>

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. Model Checking The Readers/Writers Problem

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.
26 4. Model Checking The Readers/Writers Problem

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

\text{.}
\text{.}

\text{.}

\text{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 \texttt{IsReader} is initialized with \texttt{TRUE} if the \texttt{pID} \leq n, and \texttt{FALSE} otherwise.

The transition relation is described in the \texttt{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: \texttt{index}) provides the nondeterministic choice of one process to be executed next among those processes whose corresponding guard formula is satisfied.

\[
\text{\texttt{[]}}(\texttt{[]}(p:\texttt{index}) : \texttt{c17}):
\texttt{IsReader AND s.next[pID] = r4 AND}
\texttt{s.m.cnt < 0 AND s.m.set(p)}
\rightarrow s' = (((s \texttt{WITH .m.cnt := s.m.cnt + 1}) \texttt{WITH .next[pID] := r5})
\texttt{WITH .next[p] :=}
\texttt{IF s.next[p] = waitAtPm1 THEN rlseAtPm1}
\texttt{ELSE rlseAtPm2}
\texttt{ENDIF})
\texttt{WITH .m.set := remove(p, s.m.set))}
\]

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 \texttt{process} modules and module initializator as in Figure 4.5. The result of initialization is that each process

\begin{verbatim}
main: MODULE = initializator []
   ([] (pID : index): process[pID]);
\end{verbatim}

Figure 4.5: The module main

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

\subsection{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., \texttt{semtype}) will not go over the bounds of this subrange type. This is done with the theorem \texttt{typecheck2} reproduced below:

\texttt{typecheck2: THEOREM main}
\begin{verbatim}
  |- G(s.m.cnt <= 1 AND s.m.cnt >= -M+1 AND
       s.w.cnt <= 1 AND s.w.cnt >= -M+1);
\end{verbatim}

Here, \texttt{s.m.cnt} and \texttt{s.w.cnt} are of \texttt{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:

\texttt{safety: THEOREM main}
\begin{verbatim}
  |- G((s.wt = 0 OR s.rd = 0) AND s.wt < 2
          AND s.rd >= 0 AND s.wt >= 0);
\end{verbatim}

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
4. Model Checking The Readers/Writers Problem

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:

\[
\text{dq: THEOREM main}
\]

\[
\vdash (\neg \exists (i: \text{index}) : \\
\quad \neg (G(\text{IsReader}[i] \land s.\text{next}[i] = r1 \land s.m.cnt = 1)) \\
\quad \land \neg (\exists (i: \text{index}) : \\
\quad \quad \neg (G(\text{IsReader}[i] \land s.\text{next}[i] = r1 \land s.m.cnt < 1))) \\
\quad \land \ldots \\
\quad \Rightarrow F(\forall (k: \text{index}): \text{IntRw}(s.\text{next}[k]) = 0);
\]

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 (Pos_1 > Pos_2) can be considered as the case of decreasing one of the components of the vector IntRW defined as:

\[ \text{IntRW}(\text{next}) = (\text{IntRW}(\text{next}[1]), \ldots, \text{IntRW}(\text{next}[M])) \]

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:

\[
\text{DQdecrease}(s, t: \text{state}): \text{bool} = (\text{EXISTS } (i: \text{index}): \text{IntRW}(t.\text{next}[i]) < \text{IntRW}(s.\text{next}[i])) \ \text{AND} \\
(\text{FORALL } (i: \text{index}): \\
(\text{IntRW}(t.\text{next}[i]) \leq \text{IntRW}(s.\text{next}[i]));
\]

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
\[
\neg G(\text{FORALL } (u: \text{state}): (s = u \ \text{AND} \ X(s /= u)) \\
\Rightarrow X(\text{DQdecrease}(u, s)));
\]

dqb: THEOREM main
\[
\neg AG((\text{FORALL } (t: \text{state}): (s = t \Rightarrow E\text{X}(s /= t)))) \\
\text{OR } \text{FORALL } (i: \text{index}): \text{IntRW}(s.\text{next}[i]) = 0);\]

dqc: THEOREM main
\[
\neg G((\text{FORALL } (i:\text{index}): \text{IntRW}(s.\text{next}[i]) = 0) \Rightarrow \\
\text{FORALL } (i: \text{index}): \text{s.}\text{next}[i] /= \text{waitAtPm1 OR}
\]
4. Model Checking The Readers/Writers Problem

\[
\text{s.next}[i] \neq \text{waitAtPm2} \text{ OR } \text{s.next}[i] \neq \text{waitAtPwr} \text{ OR } \\
\text{s.next}[i] \neq \text{waitAtPww});
\]

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

\text{dqmonitor : MODULE =}

\begin{verbatim}
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;
\end{verbatim}

We then verify the appropriately modified theorem:

\text{dqa\_new: THEOREM main || dqmonitor}

\[
\vdash X(G(prev_state /= s \Rightarrow \text{DQdecrease}(prev_state, s)));
\]

which is easily model-checked.

The theorem \text{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:
dqbnlp : THEOREM main || dqmonitor
|- \( X(G((\text{prev\_state} = s \Rightarrow (\exists k: \text{index}: 
\begin{align*}
&\text{IsReader}[k] \land \text{s\_next}[k] = \text{r1} \land \text{s\_m\_cnt} = 1) \lor \\
&\text{IsReader}[k] \land \text{s\_next}[k] = \text{r1} \land \text{s\_m\_cnt} < 1) \lor \\
&\ldots)) \lor \forall (i: \text{index}): \text{IntRW}(\text{s\_next}[i]) = 0))
\);

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.

<table>
<thead>
<tr>
<th>safety</th>
<th>dqa_new</th>
<th>dqbnlp</th>
<th>dqc</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>time(s)</td>
<td>states</td>
<td>time(s)</td>
</tr>
<tr>
<td>3R/2W</td>
<td>9961</td>
<td>40</td>
<td>34962</td>
</tr>
<tr>
<td>5R/5W</td>
<td>14.9·10⁶</td>
<td>2326</td>
<td>-</td>
</tr>
<tr>
<td>6R/6W</td>
<td>0.3·10⁹</td>
<td>4044</td>
<td>-</td>
</tr>
<tr>
<td>7R/7W</td>
<td>6.1·10⁹</td>
<td>55627</td>
<td>-</td>
</tr>
<tr>
<td>15R/10W</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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 \( A \rightarrow B \) 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.
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,
r: 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.

\begin{verbatim}
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
\end{verbatim}

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.

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

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

The table from [15] is modeled with a transition relation \text{trans}. The relation \text{trans}(s, t) evaluates to \text{TRUE} if one of the guard conditions \text{p1(s)} to \text{p41(s)} (whose definitions are given in Appendix C.3) holds and the program can make the transition from state \text{s} to \text{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 \text{trans} is of the type \text{stateneop}. In order to make the relation total, the first argument is subtyped to reflect the fact that some states satisfying certain predicates (\text{p1(s)}, \text{p7(s)}, \text{p10(s)}, \text{p12(s)}, \text{p15(s)}, \text{p19(s)}, \text{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]:

\[(rd = 0 \text{ or } wt = 0) \text{ and } wt < 2\]

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

\[
t: \text{VAR } \text{state}
\]

\[
\begin{align*}
rp1(t): \text{bool} &= \text{wt}(t) = 0 \text{ or } \text{rd}(t) = 0 \\
rp2(t): \text{bool} &= \text{wt}(t) < 2
\end{align*}
\]

The initial condition for the system is given by:

\[
\text{initcond}(t): \text{bool} = \text{cnt}(\text{m}(t)) = 1 \text{ and empty?}(\text{set}(\text{m}(t))) \text{ and } \\
\text{cnt}(\text{w}(t)) = 1 \text{ and empty?}(\text{set}(\text{w}(t))) \text{ and } \\
\text{rd}(t) = 0 \text{ and } \text{wt}(t) = 0 \text{ and } \text{rdcnt}(t) = 0 \text{ and } \\
(\forall (i: \text{index}): (i \leq n \text{ and } \text{next}(t)(i) = r1) \\
\text{or } (i > n \text{ and } \text{next}(t)(i) = w1))
\]

In initial state, the semaphore m is available (\text{cnt}(\text{m}(t)) = 1), and there are no processes waiting for it (\text{empty?}(\text{set}(\text{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:
40

5. Theorem Proving in PVS

\texttt{initrp1: theorem \ initcond(t) \Rightarrow 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:

\texttt{s: VAR stateneop}
\texttt{t: VAR state}
\texttt{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:

\begin{align*}
\{-1\} & \quad (\text{pID}(s!1) \leq n) \\
\{-2\} & \quad \text{rlseAtPwr?}(\text{next}(s!1)(\text{pID}(s!1))) \\
\{-3\} & \quad \text{rd}(s!1) = 0 \\
\{-4\} & \quad \text{rdcnt}(t!1) = \text{rdcnt}(s!1) \\
\{-5\} & \quad \text{rd}(t!1) = 1 \\
\{-6\} & \quad \text{wt}(t!1) = \text{wt}(s!1) \\
\{-7\} & \quad \text{cnt}(m(t!1)) = \text{cnt}(m(s!1)) \\
\{-8\} & \quad \text{set}(m(t!1)) = \text{set}(m(s!1)) \\
\{-9\} & \quad \text{cnt}(w(t!1)) = \text{cnt}(w(s!1)) \\
\{-10\} & \quad \text{set}(w(t!1)) = \text{set}(w(s!1)) \\
\{-11\} & \quad \text{pID}(t!1) = \text{pID}(s!1) \\
\{-12\} & \quad \text{r4?}(\text{next}(t!1)(\text{pID}(s!1))) \\
\{1\} & \quad \text{wt}(s!1) = 0
\end{align*}

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 \(\text{rlseAtPwr?}(\text{next}(s!1)(\text{pID}(s!1)))\) then it must be that the critical section is empty; therefore, there are no writers already writing \(\text{wt}(s!1) = 0\). The same thinking can be applied to the remaining two sequents.

Therefore, the proposed new, auxiliary invariant would be:
Theorem Proving in PVS

5. Theorem Proving in PVS

\[ \text{inv14}(t) : \text{bool} = \forall (i : \text{index}) : \text{next}(t)(i) = \text{rlseAtPwr} \]
\[ \Rightarrow \text{wt}(t) = 0 \]

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):

\[ \text{c14rp1} : \text{theorem \ } p14(s) \text{ and } rp1(s) \text{ and } V8(s) \text{ and } V10(s) \text{ and } \]
\[ \text{trans}(s, t) \implies rp1(t) \]

This theorem, however, could not be proven in PVS. The unprovable sequent indicated the \( \text{inv14} \) 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 \( \text{c14rp1} \).

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 \( (\exists i : (i = \text{pID}(t) \land \text{next}(t)(i) = l)) \Rightarrow P(t) \), 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 \( \text{pID} \) 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]:

\[ \text{V12}(t) : \text{bool} = (\exists (i : \text{index}) : i = \text{pID}(t) \land \]

\]
5. Theorem Proving in PVS

\[(\text{next}(t)(i) = r1 \text{ or next}(t)(i) = \text{rlseAtPm1} \text{ or } \\
\text{next}(t)(i) = r2 \text{ or next}(t)(i) = r4 \text{ or } \\
\text{next}(t)(i) = r5 \text{ or next}(t)(i) = r6 \text{ or } \\
\text{next}(t)(i) = \text{rlseAtPm2} \text{ or next}(t)(i) = r7 \text{ or } \\
\text{next}(t)(i) = r9) \implies \text{rd}(t) = \text{rdcnt}(t)\]

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:

\[\text{V12}(t): \text{bool} = (\exists (i: \text{index}) : \\
(\text{next}(t)(i) = r1 \text{ or next}(t)(i) = \text{rlseAtPm1} \text{ or } \\
\text{next}(t)(i) = r2 \text{ or next}(t)(i) = r4 \text{ or } \\
\text{next}(t)(i) = r5 \text{ or next}(t)(i) = r6 \text{ or } \\
\text{next}(t)(i) = \text{rlseAtPm2} \text{ or next}(t)(i) = r7 \text{ or } \\
\text{next}(t)(i) = r9) \implies \text{rd}(t) = \text{rdcnt}(t)\]

Counterexample generated by SAL is given below:

\[(r1, r1, w1, w1, 0, 0)^{pID=1} \rightarrow (r2, r1, w1, w1, 0, 0)^{pID=1} \rightarrow (r3, r1, w1, w1, 0, 1)\]

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

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]:

\[\text{V9}(t): \text{bool} = (\exists (i: \text{index}) : i = \text{pID}(t) \text{ and } \\
\text{next}(t)(i) = r7 \implies \text{rdcnt}(t) > 0\]

5. Theorem Proving in PVS

which should actually be

\[ V9_{\text{new}}(t) : \text{bool} = (\exists i: \text{index}) : \]
\[ \text{next}(t)(i) = r7 \implies \text{rdcnt}(t) > 0 \]

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 \( \text{rdcnt} \). 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:

\[ \text{CS1}(t) : \text{bool} = (\forall i, j: \text{index}) : \]
\[ \text{CS1pred}(t, i) \]
\[ \text{and} \phantom{\text{CS1pred}(t, j)} \implies i = j \]

where

\[ i : \text{VAR} \text{ index} \]
\[ \text{CS1pred}(t, i) : \text{bool} = \text{next}(t)(i) = \text{r1seAtPm1} \text{ or} \]
\[ \text{next}(t)(i) = r2 \text{ or } \text{next}(t)(i) = r3 \text{ or} \]
\[ \text{next}(t)(i) = r4 \text{ or } \text{next}(t)(i) = \text{r1seAtPm2} \text{ or} \]
\[ \text{next}(t)(i) = r7 \text{ or } \text{next}(t)(i) = r8 \text{ or} \]
\[ \text{next}(t)(i) = r9 \text{ or } \text{next}(t)(i) = \text{waitAtPwr} \text{ or} \]
\[ \text{next}(t)(i) = \text{r1seAtPwr} \]

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:

\[
\text{rp}(t) : \text{bool} = (\text{wt}(t) = 0 \text{ or } \text{rd}(t) = 0) \text{ and } \\
\text{wt}(t) < 2 \text{ and } \text{rd}(t) >= 0 \text{ and } \text{wt}(t) >= 0
\]

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

\[
\text{crp11 : lemma } \forall t: \left( \text{initcond}(t) \Rightarrow \text{rp}(t) \right) \\
\text{and } \forall s, t: \left( \left( \text{rp}(s) \text{ and } \text{trans}(s, t) \right) \Rightarrow \text{rp}(t) \right)
\]

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

\[
\begin{align*}
\text{crp11.2.1 :} & \\
{-1} & \text{pID}(t!1) \leq M \\
{-2} & \text{pID}(t!1) \geq 1 \\
{-3} & \text{r1?next}(s!1)\text{(pID}(s!1))) \\
{-4} & \text{wt}(s!1) = 0 \\
{-5} & \text{rd}(s!1) \geq 0 \\
{-6} & \text{rdcnt}(t!1) = \text{rdcnt}(s!1) \\
{-7} & \text{rd}(t!1) = \text{rd}(s!1) \\
{-8} & \text{set}(m(t!1)) = \text{set}(m(s!1)) \\
{-9} & \text{cnt}(m(t!1)) = \text{cnt}(m(s!1)) \\
{-10} & \text{cnt}(w(t!1)) = 1 + \text{cnt}(w(s!1)) \\
{-11} & \text{set}(w(s!1))(pID(t!1)) \\
{-12} & \text{set}(w(t!1)) = \text{remove}(pID(t!1), \text{set}(w(s!1))) \\
{-13} & \text{wt}(t!1) = -1 \\
{-14} & (p!1 = \text{pID}(t!1))
\end{align*}
\]
Theorem Proving in PVS

\[ \text{waitAtPww?}(\text{next}(s!1)(\text{pID}(t!1))) \]
\[ \text{rlseAtPww?}(\text{next}(t!1)(\text{pID}(t!1))) \]

\[ \begin{align*}
\{1\} & \quad \text{pID}(s!1) > n \\
\{2\} & \quad \text{cnt}(m(s!1)) = 1 \\
\{3\} & \quad \text{EOO?}(\text{next}(t!1)(\text{pID}(t!1))) \\
\{4\} & \quad \text{cnt}(m(s!1)) < 1 \\
\{5\} & \quad (\text{pID}(t!1) = \text{pID}(s!1))
\end{align*} \]

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

\(S1(t): \text{bool} = \text{cnt}(m(t)) \leq 1\)

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:

\(s: \text{VAR stateneop}\)
\(t: \text{VAR state}\)
\(\text{ind1}(t): \text{bool} = \text{rp}(t) \land S1(t) \land S2(t) \land S31(t)\)
\(\quad \land S32(t) \land S41(t) \land S5(t)\)
\(\quad \land S6(t) \land S7(t) \land S81(t)\)
\(\quad \land S82(t) \land S91(t) \land S101(t)\)
\(\text{crpind1: lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{ind1}(t))\)
\(\quad \land (\forall s, t: \text{trans}(s, t) \Rightarrow \text{ind1}(t))\)

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 \(i!1 = \text{pID}(s!1)\), we apply \text{GRIND}, and contemplate
the invariants from the unprovable sequents. However, we choose to skip the case for $i!1 \neq \text{pID}(s!1)$, 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 ($i!1 \neq \text{pID}(s!1)$) is the following:

$$\text{crpind1.2.4.2.2 :}$$

\begin{itemize}
  \item [-1] $r4?(\text{next}(s!1)(i!1))$
  \item [-2] $r4?(\text{next}(s!1)(\text{pID}(s!1)))$
  \item [-3] $wt(s!1) = 0$
  \item [-4] $rd(s!1) >= 0$
  \item [-5] $(\text{cnt}(w(s!1)) <= 1)$
  \item [-6] $(\text{rdcnt}(s!1)) >= 0)$
  \item [-7] $\text{cnt}(m(s!1)) = 0$
  \item [-8] $\text{rdcnt}(t!1) = \text{rdcnt}(s!1)$
  \item [-9] $\text{rd}(t!1) = \text{rd}(s!1)$
  \item [-10] $wt(t!1) = 0$
  \item [-11] $\text{cnt}(m(t!1)) = 1$
  \item [-12] $\text{set}(m(t!1)) = \text{set}(m(s!1))$
  \item [-13] $\text{cnt}(w(t!1)) = \text{cnt}(w(s!1))$
  \item [-14] $\text{set}(w(t!1)) = \text{set}(w(s!1))$
  \item [-15] $\text{next}(t!1)(i!1) = \text{next}(s!1)(i!1)$
\end{itemize}

\[i!1 = \text{pID}(s!1)\]

\[r1?(\text{next}(s!1)(\text{pID}(s!1)))\]

\[\text{waitAtPm1?(next}(s!1)(\text{pID}(s!1))\right)\]

\[\text{rlseAtPm1?(next}(s!1)(\text{pID}(s!1))\right)\]

\[r2?(\text{next}(s!1)(\text{pID}(s!1)))\]

\[r3?(\text{next}(s!1)(\text{pID}(s!1)))\]

\[\text{waitAtPwr?}(\text{next}(s!1)(\text{pID}(s!1)))\]

\[\text{rlseAtPwr?}(\text{next}(s!1)(\text{pID}(s!1)))\]

\[\text{pID}(s!1) > n\]

\[\text{cnt}(w(s!1)) = 1\]

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.
5. Theorem Proving in PVS

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 \( i \neq 1 \). 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 \( i \neq 1 \)) 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:

```plaintext
% 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)));```
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:

\[
\text{s:VAR stateneop} \\
\text{t: VAR state} \\
\text{DQdecrease(s, t): bool = (exists i: IntRW(next(s)(i)) > IntRW(next(t)(i))) and} \\
\text{ (forall i: IntRW(next(s)(i)) >= IntRW(next(t)(i)))}
\]

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):

\[
\text{s:VAR stateneop} \\
\text{t: VAR state} \\
\text{dqa: theorem indc4(s) => (trans(s, t) and not (m(s) = m(t) and} \\
\text{ w(s) = w(t) and rdcnt(s) = rdcnt(t) and}
\]

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.
Theorem Proving in PVS

\[
\text{(forall } i: \text{next}(s)(i) = \text{next}(t)(i)) \text{ and } \\
\text{rd}(s) = \text{rd}(t) \text{ and } \text{wt}(s) = \text{wt}(t) \Rightarrow DQdecrease(s, t))
\]

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 \text{pID}). We formalize it as:

\[
s1, t, u: \text{VAR state} \\
dqb: \text{lemma } \text{forall } s1: (\text{indc4}(s1) \Rightarrow \\
(\text{forall } i: \text{IntRW}(\text{next}(s1)(i)) = 0) \text{ or } \\
(\exists t: (\text{trans}(s1, t) \text{ and } \text{not } (m(s1) = m(t) \text{ and } \text{w}(s1) = w(t) \text{ and } \text{rdcnt}(s1) = \text{rdcnt}(t) \text{ and } \\
(\forall i: \text{next}(s1)(i) = \text{next}(t)(i)) \text{ and } \\
\text{rd}(s1) = \text{rd}(t) \text{ and } \text{wt}(s1) = \text{wt}(t)) \text{ or } \\
(\exists u: (\text{trans}(t, u) \text{ and } \text{not } (m(t) = m(u) \text{ and } \text{w}(t) = w(u) \text{ and } \text{rdcnt}(t) = \text{rdcnt}(u) \text{ and } \\
(\forall i: \text{next}(t)(i) = \text{next}(u)(i)) \text{ and } \\
\text{rd}(t) = \text{rd}(u) \text{ and } \text{wt}(t) = \text{wt}(u))))))))))
\]

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

\[
t: \text{VAR state} \\
dqc: \text{theorem } \text{indc4}(t) \Rightarrow (\forall i: \text{index}: \text{IntRW}(\text{next}(t)(i)) = 0) \\
\text{implies } (\forall i: \text{index}: (\text{next}(t)(i) /= \text{waitAtPm1} \\
\text{and } \text{next}(t)(i) /= \text{waitAtPm2} \text{ and } \text{next}(t)(i) /= \text{waitAtPwr} \text{ and } \text{next}(t)(i) /= \text{waitAtPww}))
\]

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
6. Conclusion

of the form $\forall i : (next(t)) = l \Rightarrow P(t)$, 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


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.

<table>
<thead>
<tr>
<th>sem.cnt &gt; 1</th>
<th>sem.cnt = 1</th>
<th>sem.cnt &lt; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem.cnt’</td>
<td>false</td>
<td>sem.cnt’ = ‘sem.cnt − 1</td>
</tr>
<tr>
<td>sem.set’</td>
<td>false</td>
<td>sem.cnt’ = ‘sem.set</td>
</tr>
<tr>
<td>next’</td>
<td>false</td>
<td>Table a)</td>
</tr>
</tbody>
</table>

Table a): ∀j, j = i
next[j]’ = NextLabel(x) next[j]’ = next[j]
j ≠ i

Table b): ∀j, j = i
next[j]’ = waitAtPsem next[j]’ = ‘next[j]’
j ≠ i

Figure A.1: Specification of P(sem) operation
A. Specification of P/V Semaphore Operations

<table>
<thead>
<tr>
<th></th>
<th>'sem.cnt &gt; 0'</th>
<th>'sem.cnt = 0'</th>
<th>'sem.cnt &lt; 0'</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem.cnt'</td>
<td>false</td>
<td>sem.cnt' = 'sem.cnt + 1</td>
<td>sem.cnt' = 'sem.cnt + 1</td>
</tr>
<tr>
<td>sem.set'</td>
<td>false</td>
<td>sem.cnt' = 'sem.set'</td>
<td>(\exists t : (t \in \text{sem.set} \land \text{sem.set}' = \text{sem.set} - {t}))</td>
</tr>
<tr>
<td>next'</td>
<td>false</td>
<td>Table a)</td>
<td>Table b)</td>
</tr>
</tbody>
</table>

Table a): \(\forall j,\)

\[
\begin{array}{c|c}
  j = i & j \neq i \\
  next[j]' = NextLabel(x) & next[j]' = 'next[j]\end{array}
\]

Table b): \(\forall j,\)

\[
\begin{array}{c|c|c|c}
  j = i & j \neq i \land j \in ('\text{sem.set} - \text{sem.set}') \land 'next[j]' = \text{sem.set}' & 'next[j]' = \text{waitAtPsem}
  next[j]' = NextLabel(x) & next[j]' = 'next[j]\end{array}
\]

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.
\[ pID = k \land IsReader \]

<table>
<thead>
<tr>
<th>( m.cnt \geq 1 )</th>
<th>( m.cnt = 0 )</th>
<th>( m.cnt &lt; 0 )</th>
<th>( \text{waitAtPm1} )</th>
<th>( \text{riseAtPm1} )</th>
<th>( \text{waitAtPm2} )</th>
<th>( \text{riseAtPm2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{next}[k] = r1 )</td>
<td>( \text{next}[k] = r2 )</td>
<td>( \text{next}[k] = r3 )</td>
<td>( \text{next}[k] = r4 )</td>
<td>( \text{next}[k] = r5 )</td>
<td>( \text{next}[k] = r6 )</td>
<td></td>
</tr>
<tr>
<td>( \text{rdcnt} \geq 1 )</td>
<td>( \text{rdcnt} = 1 )</td>
<td>( \text{rdcnt} &gt; 1 )</td>
<td>( \text{rdcnt} \geq 1 )</td>
<td>( \text{rdcnt} = 1 )</td>
<td>( \text{rdcnt} &gt; 1 )</td>
<td></td>
</tr>
<tr>
<td>( \text{rdcnt} )</td>
<td>( \text{rdcnt} )</td>
<td>( \text{rdcnt} )</td>
<td>( \text{rdcnt} )</td>
<td>( \text{rdcnt} )</td>
<td>( \text{rdcnt} )</td>
<td></td>
</tr>
<tr>
<td>( \text{rd} )</td>
<td>( \text{rd} )</td>
<td>( \text{rd} )</td>
<td>( \text{rd} )</td>
<td>( \text{rd} )</td>
<td>( \text{rd} )</td>
<td></td>
</tr>
<tr>
<td>( \text{set} )</td>
<td>( \text{set} )</td>
<td>( \text{set} )</td>
<td>( \text{set} )</td>
<td>( \text{set} )</td>
<td>( \text{set} )</td>
<td></td>
</tr>
<tr>
<td>( m.cnt )</td>
<td>( m.cnt )</td>
<td>( m.cnt )</td>
<td>( m.cnt )</td>
<td>( m.cnt )</td>
<td>( m.cnt )</td>
<td></td>
</tr>
<tr>
<td>( w.cnt )</td>
<td>( w.cnt )</td>
<td>( w.cnt )</td>
<td>( w.cnt )</td>
<td>( w.cnt )</td>
<td>( w.cnt )</td>
<td></td>
</tr>
<tr>
<td>( pID' )</td>
<td>( \text{next}[pID'] \neq \text{next}[pID'] )</td>
<td>( \text{next}[pID'] \neq \text{next}[pID'] )</td>
<td>( \text{next}[pID'] \neq \text{next}[pID'] )</td>
<td>( \text{next}[pID'] \neq \text{next}[pID'] )</td>
<td>( \text{next}[pID'] \neq \text{next}[pID'] )</td>
<td></td>
</tr>
</tbody>
</table>

Figure B.1: The tabular representation of the rewritten Readers/Writers Program
<table>
<thead>
<tr>
<th>$\text{next}[k] = rT$</th>
<th>$\text{next}[k] = r8$</th>
<th>$\text{next}[k] = r9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rdent} = \text{rdent} + 1$</td>
<td>$\text{rdent} &gt; 0$</td>
<td>$\text{rdent} &lt; 0$</td>
</tr>
<tr>
<td>$\text{w.cnt} &gt; 0$</td>
<td>$\text{w.cnt} &lt; 0$</td>
<td>$\text{m.cnt} &gt; 0$</td>
</tr>
<tr>
<td>$\text{m.cnt} = 0$</td>
<td>$\text{m.cnt} &lt; 0$</td>
<td>$\text{m.cnt} &lt; 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{rd} = \text{rd}$</th>
<th>$\text{rd} &gt; 0$</th>
<th>$\text{rd} &lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rd} = \text{rd} - 1$</td>
<td>$\text{rd} &gt; 0$</td>
<td>$\text{rd} &lt; 0$</td>
</tr>
<tr>
<td>$\text{rd} = \text{rd}$</td>
<td>$\text{rd} &gt; 0$</td>
<td>$\text{rd} &lt; 0$</td>
</tr>
<tr>
<td>$\text{wt} = \text{wt}$</td>
<td>$\text{wt} &gt; 0$</td>
<td>$\text{wt} &lt; 0$</td>
</tr>
<tr>
<td>$\text{m.cnt} = \text{m.cnt}$</td>
<td>$\text{m.cnt} &gt; 0$</td>
<td>$\text{m.cnt} &lt; 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{w.set} = \text{w.set}$</th>
<th>$\text{w.set} &gt; 0$</th>
<th>$\text{w.set} &lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{w.set} = \text{w.set} - \text{wt}$</td>
<td>$\text{w.set} &gt; 0$</td>
<td>$\text{w.set} &lt; 0$</td>
</tr>
<tr>
<td>$\text{w.set} = \text{w.set}$</td>
<td>$\text{w.set} &gt; 0$</td>
<td>$\text{w.set} &lt; 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{pID}' \neq \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
<th>$\text{next}[\text{pID}'] = \text{EOP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{next}[\text{pID}'] = \text{next}[\text{pID}']$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
<td>$\text{next}[\text{pID}'] = \text{EOP}$</td>
</tr>
</tbody>
</table>

Figure B.2: Figure B.1 continued
### B. The Tabular Representation of the Rewritten Readers/Writers

#### Program

| Tab2: ∀j,  | j = k | j ≠ k | next[j]' = r2 | next[j]' = 'next[j] |
| Tab3: ∀j,  | j = k | j ≠ k | next[j]' = waitAtPm1 | next[j]' = 'next[j] |
| Tab4: ∀j,  | j = k | j ≠ k | next[j]' = 'next[j] | next[j]' = 'next[j] |
| Tab5: ∀j,  | j = k | j ≠ k | next[j]' = r2 | next[j]' = 'next[j] |
| Tab6: ∀j,  | j = k | j ≠ k | next[j]' = r3 | next[j]' = 'next[j] |
| Tab7: ∀j,  | j = k | j ≠ k | next[j]' = waitAtPwr | next[j]' = 'next[j] |
| Tab8: ∀j,  | j = k | j ≠ k | next[j]' = r4 | next[j]' = 'next[j] |
| Tab9: ∀j,  | j = k | j ≠ k | next[j]' = 'next[j] | next[j]' = 'next[j] |
| Tab10: ∀j, | j = k | j ≠ k | next[j]' = r5 | next[j]' = 'next[j] |
| Tab11: ∀j,| j = k | j ≠ k | next[j]' = r6 | next[j]' = 'next[j] |
| Tab12: ∀j,| j = k | j ≠ k | next[j]' = r7 | next[j]' = 'next[j] |
| Tab13: ∀j,| j = k | j ≠ k | next[j]' = waitAtPwr | next[j]' = 'next[j] |
| Tab14: ∀j,| j = k | j ≠ k | next[j]' = r8 | next[j]' = 'next[j] |
| Tab15: ∀j,| j = k | j ≠ k | next[j]' = r9 | next[j]' = 'next[j] |
| Tab16: ∀j,| j = k | j ≠ k | next[j]' = waitAtPwr | next[j]' = 'next[j] |
| Tab17: ∀j,| j = k | j ≠ k | next[j]' = r9 | next[j]' = 'next[j] |
| Tab18: ∀j,| j = k | j ≠ k | next[j]' = waitAtPm2 | next[j]' = 'next[j] |
| Tab19: ∀j,| j = k | j ≠ k | next[j]' = r9 | next[j]' = 'next[j] |
| Tab20: ∀j,| j = k | j ≠ k | next[j]' = waitAtPm2 | next[j]' = 'next[j] |
| Tab21: ∀j,| j = k | j ≠ k | next[j]' = r9 | next[j]' = 'next[j] |
| Tab22: ∀j,| j = k | j ≠ k | next[j]' = waitAtPm2 | next[j]' = 'next[j] |
| Tab23: ∀j,| j = k | j ≠ k | next[j]' = r9 | next[j]' = 'next[j] |
| Tab24: ∀j,| j = k | j ≠ k | next[j]' = r9 | next[j]' = 'next[j] |
B. The Tabular Representation of the Rewritten Readers/Writers Program

<table>
<thead>
<tr>
<th>Tab26: ∀j, j = k</th>
<th>j ≠ k</th>
<th>next[j]' = r9</th>
<th>next[j]' = waitAtPww</th>
<th>next[j]' = rlseAtPww</th>
<th>next[j]' = wa't next[j]</th>
</tr>
</thead>
</table>

| Tab27: ∀j, j = k | j ≠ k ∧ j ∈ \('m.set - m.set'\) | \(j ≠ k ∧ (j \notin \('m.set - m.set'\) ∨ \('next[j] = waitAtPww\) \lor \neg\(next[j] = waitAtPww\)) \) | next[j]' = r9 | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = wa't next[j] |

| Tab29: ∀j, j = k | j ≠ k | next[j]' = r9 | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = wa't next[j] |

| Tab31: ∀j, j = k | j ≠ k | next[j]' = EOP | next[j]' = wa't next[j] |

| Tab32: ∀j, j = k | j ≠ k | next[j]' = r9 | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = wa't next[j] |

| Tab34: ∀j, j = k | j ≠ k | next[j]' = w2 | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = wa't next[j] |

| Tab35: ∀j, j = k | j ≠ k | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = EOP | next[j]' = wa't next[j] |

| Tab36: ∀j, j = k | j ≠ k | next[j]' = next[j] | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = wa't next[j] |

| Tab37: ∀j, j = k | j ≠ k | next[j]' = w2 | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = EOP | next[j]' = wa't next[j] |

| Tab38: ∀j, j = k | j ≠ k | next[j]' = w3 | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = EOP | next[j]' = wa't next[j] |

| Tab40: ∀j, j = k | j ≠ k | next[j]' = EOP | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = w2 | next[j]' = wa't next[j] |

| Tab41: ∀j, j = k | j ≠ k ∧ j ∈ \('w.set - w.set'\) | \(j ≠ k ∧ (j \notin \('w.set - w.set'\) ∨ \('next[j] = waitAtPww\) \lor \neg\(next[j] = waitAtPww\)) \) | next[j]' = EOP | next[j]' = waitAtPww | next[j]' = rlseAtPww | next[j]' = wa't next[j] |
Appendix C

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

C.1 The Readers/Writers Model in SPIN

```plaintext
/*The Readers/Writers Model in SPIN*/

struct {p, v};
chan mutex = [0] of {struct};
chan w = [0] of {struct};
int wt, rd, rdcnt = 0;
active proctype mi()
{
    byte count=!;
    do :: (count = 1) ->
    end: mutex?p; count = 0
    :: (count = 0) ->
    mutex?v; count = 1
    od
}
active proctype m2()
{
    byte count=!;
    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 ->
        f1;
    rd++
    mutex?v;
```
C. The Readers/Writers Model in SPIN, SAL, and PVS

C.2 Model of Readers/Writers Program in SAL

```plaintext
rws3(); M : nnat, n : nat): CONTEXT =
BEGIN
  Job_idx: TYPE = [1..M];
  label: TYPE = [r1, waitAtPal, r3AtPal, r3, r3, waitAtPwr, r3AtPwr,
  r4, r5, r5, waitAtPw2, r3AtPw2, r7, r8, r9, w1, w2, w3,
  waitAtPv, r3AtPv, EOP];
  rdtypa: TYPE = [-1..n+1];
  wttypa: TYPE = [-1..(M-n+1)];
  xntypa: TYPE = [1..M];
  index: TYPE = [1..M];
  setof: TYPE = [index -> bool];
  member(x: index, a: setof): bool = a(x);
  empty(x: setof): bool = (FORALL x: index) NOT member(x, a);
  emptysetof = (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 #];
  IntN(x: label): int =
  IF x=r1 THEN 15
  ELIF x=r1AtPal THEN
  ELIF x=r1AtPal THEN 13
  ELIF x=r1 THEN 11
  ELIF x=r1AttPw THEN 10
  ELIF x=r1AtPv THEN 9
  ELIF x=r1 THEN 8
  ELIF x=r1 THEN 7
  ELIF x=r1 THEN 6
  ELIF x=r1AttPw THEN 5
  ELIF x=r1AtPw THEN 4
  ELIF x=r1 THEN 3
  ELIF x=r1 THEN 2
  ELIF x=r1 THEN 1
  ELIF x=r1 THEN 5
  ELIF x=r1AttPw THEN 4
  ELIF x=r1AtPw THEN 3
  ELIF x=r1 THEN 2
```
ENDIF;

state: TYPE = {#
  s: sem,
  v: sem,
  rdct: 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

{ c3:
  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)
}

{ c5:
  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
}

{ c6:
  IsReader AND s.next[PID] = waitAtPm1
  --> s' = s
}

{ c4:
  IsReader AND s.next[PID] = rnextAtPm1
  --> s' = ((s WITH .next[PID] := r2)
}

{ c7:
  IsReader AND s.next[PID] = r2
  --> s' = ((s WITH .rdcat := s.rdcat + 1)
  WITH .next[PID] := r3
}

{ 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
}
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[\text{IsReader AND s.next[piD] = r3 AND s.rdct < 1} \Rightarrow \]
\[\text{\{\}}\]
\[\text{c11:}\]
\[\text{IsReader AND s.next[piD] = r3 AND s.rdct > 1 AND s.w.cnt < 1} \Rightarrow \]
\[s' = (s \text{ WITH .rd} := s.rd + 1) \text{ WITH .next[piD]} := r4\]
\[\text{\{\}}\]
\[\text{c12:}\]
\[\text{IsReader AND s.next[piD] = r3 AND s.rdct > 1 AND s.w.cnt >= 1} \Rightarrow \]
\[\text{\{\}}\]
\[\text{c13:}\]
\[\text{IsReader AND s.next[piD] = waitAtPwr} \Rightarrow \]
\[s' = s\]
\[\text{\{\}}\]
\[\text{c14:}\]
\[\text{IsReader AND s.next[piD] = riseAtPwr} \Rightarrow \]
\[s' = (s \text{ WITH .rd} := s.rd + 1) \text{ WITH .next[piD]} := r4\]
\[\text{\{\}}\]
\[\text{c15:}\]
\[\text{IsReader AND s.next[piD] = r4 AND s.w.cnt > 0} \Rightarrow \]
\[\text{\{\}}\]
\[\text{c16:}\]
\[\text{IsReader AND s.next[piD] = r4 AND s.w.cnt = 0} \Rightarrow \]
\[s' = (s \text{ WITH .m.cnt} := s.m.cnt + 1) \text{ WITH .next[piD]} := r6\]
\[\text{\{\}}\]
\[\text{(p: index):}\]
\[\text{c17:}\]
\[\text{IsReader AND s.next[piD] = r4 AND s.m.cnt < 0 AND s.m.set(p) \Rightarrow s' = ((s \text{ WITH .m.cnt} := s.m.cnt + 1)} \text{ WITH .next[piD]} := r6\)
\[\text{WITH .next[p] := IF s.next[p] = waitAtPwr THEN riseAtPwr ELSE riseAtPwr ELSE}}\]
\[\text{ENDIF) \text{ WITH .m.set := remove(p, s.m.set))} \text{WITH .next[p]} := r6\]
\[\text{\{\}}\]
\[\text{c18:}\]
\[\text{IsReader AND s.next[piD] = r6} \Rightarrow \]
\[s' = (s \text{ WITH .next[piD]} := r6)\]
\[\text{\{\}}\]
\[\text{c19:}\]
\[\text{IsReader AND s.next[piD] = r6 AND s.m.cnt > 1} \Rightarrow \]
\[\text{\{\}}\]
\[\text{c20:}\]
\[\text{IsReader AND s.next[piD] = r6 AND s.m.cnt = 1} \Rightarrow \]
\[s' = (s \text{ WITH .m.cnt} := s.m.cnt - 1) \text{ WITH .next[piD]} := r7\]
\[\text{\{\}}\]
\[\text{c21:}\]
\[\text{IsReader AND s.next[piD] = r6 AND s.m.cnt < 1} \Rightarrow \]
\[s' = (s \text{ WITH .m.cnt} := s.m.cnt - 1) \text{ WITH .m.set} := \text{union}(x: \text{ index x} = \text{piD}, s.m.set)) \text{ WITH .next[piD]} := \text{waitAtPwr}\]
\[\text{\{\}}\]
\[\text{c22:}\]
\[\text{IsReader AND s.next[piD] = waitAtPwr} \Rightarrow \]
\[s' = s\]
\[\text{\{\}}\]
\[\text{c23:}\]
\[\text{IsReader AND s.next[piD] = riseAtPwr}\]
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[
\begin{align*}
\text{\texttt{c24:}} & \quad \text{IsReader AND s.next[PID] = r7} \\
& \quad \to s' = s \text{ WITH .next[PID] := r7} \\
\text{\texttt{c25:}} & \quad \text{IsReader AND s.next[PID] = r8 AND s.rdcnt = 0 AND s.w.cnt > 0} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r8}) \\
\text{\texttt{c26:}} & \quad \text{IsReader AND s.next[PID] = r9 AND s.rdcnt = 0 AND s.w.cnt = 0} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r9}) \\
\text{\texttt{c27:}} & \quad \text{IsReader AND s.next[PID] = r8 AND s.rdcnt = 0} \\
& \quad \text{AND s.w.cnt < 0 AND s.w.set(p) AND s.next(p) = waitAtPmv} \\
& \quad \to s' = (((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r8}) \text{ WITH .w.cnt := s.w.cnt + 1}) \\
& \quad \text{WITH .w.set := remove(p, s.w.set)}) \\
\text{\texttt{c28:}} & \quad \text{IsReader AND s.next[PID] = r9 AND s.rdcnt < 0} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r9}) \\
\text{\texttt{c29:}} & \quad \text{IsReader AND s.next[PID] = r8 AND s.rdcnt > 0} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r8}) \\
\text{\texttt{c30:}} & \quad \text{IsReader AND s.next[PID] = r9 AND s.w.cnt > 0} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r9}) \\
\text{\texttt{c31:}} & \quad \text{IsReader AND s.next[PID] = r9 AND s.w.cnt = 0} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r9}) \\
\text{\texttt{c32:}} & \quad \text{IsReader AND s.next[PID] = r9 AND} \\
& \quad \text{s.w.cnt < 0 AND s.w.set(p)} \\
& \quad \to s' = (((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .next[PID] := r9}) \text{ WITH .w.cnt := s.w.cnt + 1}) \\
& \quad \text{WITH .w.set := remove(p, s.w.set)}) \\
\text{\texttt{c33:}} & \quad \text{NOT IsReader AND s.next[PID] = w1 AND s.w.cnt > 1} \\
& \quad \to s' = ((s \text{ WITH .rd := s.rd - 1}) \text{ WITH .w.cnt := s.w.cnt - 1}) \\
& \quad \text{WITH .next[PID] := w2}
\end{align*}
\]
C. The Readers/Writers Model in SPIN, SAL, and PVS

```plaintext
[]
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] := waitAtPw
[]
c36:
  NOT IsReader AND s.next[pID] = waitAtPw
  --> s' = s
[]
c37:
  NOT IsReader AND s.next[pID] = riseAtPw
  --> s' = (s WITH .w := s.w + 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 .w := s.w - 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 .w := s.w - 1)
  WITH .next[pID] := EOP)
WITH .w.cnt := s.w.cnt + 1)
WITH .next[pID] := EOP
ELSE riseAtPw
ENDIF
WITH .w.set := remove(p, s.w.set))
]
END;
dqmonitor : MODULE =
BEGIN
INPUT s : state
OUTPUT prev_state : state
INITIALIZATION
prev_state = (((# s := (# cnt := 1, set := (x:index | false) #),
w := (# cnt := 1, set := (x: index | false) #),
rdcnt := 0, next := [1: index] IF 1 <= n THEN r1
ELSE w1
ENDIF),
rd := 0, wt := 0 #)
TRANSITION
prev_state' = s;
END;
Initializator: MODULE =
BEGIN
GLOBAL s: state
INITIALIZATION
s = (((# s := (# cnt := 1, set := (x:index | false) #),
  w := (# cnt := 1, set := (x: index | false) #),
  rdcnt := 0, next := [1: index] IF 1 <= n THEN r1
  ELSE w1
ENDIF),
rd := 0, wt := 0 #)
TRANSITION
prev_state' = s;
END;
```
C. The Readers/Writers Model in SPIN, SAL, and PVS

C.3 PVS files

1. decl.pvs

```
dcl: THEORY

BEGIN
NM 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, r2, r3, waitAtPvr, r3AtPvr, r4, r5, r6, waitAtPm2, r3AtPm2,
r7, r8, r9, w1, w2, w3, waitAtPw, r3AtPw, EOP}
x: VAR label
rlabel?(x): bool = (x = r1 or x = waitAtPm1 or
x = r2AtPm1 or x = r2 or
x = r3 or x = waitAtPvr or
x = r3AtPvr or x = r4 or
x = r5 or x = r6 or
x = waitAtPm2 or x = r3AtPm2 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 = waitAtPw or x = r3AtPw or
x = EOP)

Now 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):
((1 <= n -> rlabel?(a(i))) and
(1 > n -> wlabel?(a(i)))))

IsReader(i: index): bool = (1 <= n)

IntRW(x: int) = int

COND
x=r1 ->15,
x=waitAtPm1 -> 14,
x=r2AtPm1 ->13,
x=r2 ->12,
x=waitAtPw ->11,
x=r3AtPw ->10,
x=r3AtPm2 ->9,
x=r4 ->8,
x=r5 ->7,
x=r6 ->6,
x=waitAtPw2 ->5,
x=r3AtPw2 ->4,
x=r7 -> 3,
x=r8 ->2,
x=r9 ->1,
x=waitAtPm2 ->0,
```
C. The Readers/Writers Model in SPIN, SAL, and PVS

2. conds.pvs

conds: THEORY

BEGIN
importing decl
s: VAR statenecop
p1(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1 and cnt(s(a)) > 1
p2(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1 and cnt(s(a)) = 1
p3(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1 and cnt(s(a)) < 1
p4(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = wai11AtPw1
p5(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1sAtPw1
p6(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r2
p7(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r3 and rdctn(s) = 1 and cnt(w(s)) > 1
p8(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r3 and rdctn(s) = 1 and cnt(w(s)) = 1
p9(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r3 and rdctn(s) < 1
p10(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r3 and rdctn(s) < 1
p11(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r3 and rdctn(s) > 1 and cnt(w(s)) < 1
p12(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r3 and rdctn(s) > 1 and cnt(w(s)) = 1
p13(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = wai11AtPwP
p14(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1sAtPw1
p15(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r4 and cnt(s(a)) > 0
p16(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r4 and cnt(s(a)) = 0
p17(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r4 and cnt(s(a)) < 0
p18(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r6
p19(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r6 and cnt(s(a)) > 1
p20(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r6 and cnt(s(a)) = 1
p21(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r6 and cnt(s(a)) < 1
p22(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1sAtPw2
p23(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r1sAtPw2
p24(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r7
p25(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r8 and rdctn(s) = 0 and cnt(w(s)) > 0
p26(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r8 and rdctn(s) = 0 and cnt(w(s)) = 0
p27(a): bool = IsHeader(pID(a)) and next(a)(pID(a)) = r8 and rdctn(s) = 0 and cnt(w(s)) < 0
C. The Readers/Writers Model in SPIN, SAL, and PVS

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(w(s)) > 0
p31(s): bool = IsReader(piD(s)) and next(s)(piD(s)) = r9 and cnt(w(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)) = w2 and cnt(w(s)) < 1
p36(s): bool = not IsReader(piD(s)) and next(s)(piD(s)) = waitAtPvw
p37(s): bool = not IsReader(piD(s)) and next(s)(piD(s)) = rleasetPvw
p38(s): bool = not IsReader(piD(s)) and next(s)(piD(s)) = w3 and cnt(w(s)) > 0
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

3. table.pvs

transition % [ parameters ] : THEORY
BEGIN
importing conde
j: VAR index
trans(s : {s : statenope
 NOT (p1(s) or p7(s) or
 p10(s) or p12(s)
 or p15(a) 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 Y.------------------------------------------------------------11
lp1(s)
------------------------------------------------------------11
Y.------------------------------------------------------------11
lp2(a)
------------------------------------------------------------11
Y.------------------------------------------------------------11
lp3(s)
------------------------------------------------------------11
Y.------------------------------------------------------------11
lp4(s)
------------------------------------------------------------11
Y.------------------------------------------------------------11
lp5(s)
------------------------------------------------------------11
Y.------------------------------------------------------------11
lp6(s)
------------------------------------------------------------11

table

<table>
<thead>
<tr>
<th>transition</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>p28(s)</td>
<td>bool = IsReader(piD(s)) and next(s)(piD(s)) = r8 and rdcnt(s) &lt; 0</td>
</tr>
<tr>
<td>p29(s)</td>
<td>bool = IsReader(piD(s)) and next(s)(piD(s)) = r8 and rdcnt(s) &gt; 0</td>
</tr>
<tr>
<td>p30(s)</td>
<td>bool = IsReader(piD(s)) and next(s)(piD(s)) = r9 and cnt(w(s)) &gt; 0</td>
</tr>
<tr>
<td>p31(s)</td>
<td>bool = IsReader(piD(s)) and next(s)(piD(s)) = r9 and cnt(w(s)) = 0</td>
</tr>
<tr>
<td>p32(s)</td>
<td>bool = IsReader(piD(s)) and next(s)(piD(s)) = r9 and cnt(m(s)) &lt; 0</td>
</tr>
<tr>
<td>p33(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = w1 and cnt(w(s)) &lt; 1</td>
</tr>
<tr>
<td>p34(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = w1 and cnt(w(s)) = 1</td>
</tr>
<tr>
<td>p35(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = w2 and cnt(w(s)) &lt; 1</td>
</tr>
<tr>
<td>p36(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = waitAtPvw</td>
</tr>
<tr>
<td>p37(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = rleasetPvw</td>
</tr>
<tr>
<td>p38(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = w3 and cnt(w(s)) &gt; 0</td>
</tr>
<tr>
<td>p39(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = w3 and cnt(w(s)) = 0</td>
</tr>
<tr>
<td>p40(s)</td>
<td>bool = not IsReader(piD(s)) and next(s)(piD(s)) = w3 and cnt(w(s)) &lt; 0</td>
</tr>
</tbody>
</table>

END conde
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[ \text{set}(a(t)) = \text{set}(a(s)) \text{ and } \text{cnt}(w(t)) = \text{cnt}(w(s)) \text{ and } \]
\[ \text{set}(v(t)) = \text{set}(v(s)) \text{ and } \]
\[ (\forall j: (j \neq k \text{ and next}(t)(j) = r3) \text{ or } (j \neq k \text{ and next}(t)(j) = next(s)(j))) \text{ and } \]
\[ \text{next}(t)(\text{PID}(t)) \rightarrow EDP \]

\[ \text{ip7}(a) \]

\[ \text{ip8}(a) | \text{rdcnt}(t) = \text{rdcnt}(a) \text{ and } \text{rd}(t) = \text{rd}(a) + 1 \text{ and } \]
\[ \text{wt}(t) = \text{wt}(s) \text{ and } \text{cnt}(w(t)) = \text{cnt}(w(s)) \text{ and } \]
\[ \text{set}(w(t)) = \text{set}(w(s)) \text{ and } \]
\[ (\forall j: (j \neq k \text{ and next}(t)(j) = r4) \text{ or } (j \neq k \text{ and next}(t)(j) = next(s)(j))) \text{ and } \]
\[ \text{next}(t)(\text{PID}(t)) \rightarrow EDP \]

\[ \text{ip9}(a) \]

\[ \text{ip10}(a) \]

\[ \text{ip11}(a) | \text{rdcnt}(t) = \text{rdcnt}(a) \text{ and } \text{rd}(t) = \text{rd}(a) + 1 \text{ and } \]
\[ \text{wt}(t) = \text{wt}(s) \text{ and } \text{cnt}(w(t)) = \text{cnt}(w(s)) \text{ and } \]
\[ \text{set}(w(t)) = \text{set}(w(s)) \text{ and } \]
\[ (\forall j: (j \neq k \text{ and next}(t)(j) = r4) \text{ or } (j \neq k \text{ and next}(t)(j) = next(s)(j))) \text{ and } \]
\[ \text{next}(t)(\text{PID}(t)) \rightarrow EDP \]

\[ \text{ip12}(a) \]

\[ \text{ip13}(a) \]

\[ \text{ip14}(a) | \text{rdcnt}(t) = \text{rdcnt}(a) \text{ and } \text{rd}(t) = \text{rd}(a) + 1 \text{ and } \]
\[ \text{wt}(t) = \text{wt}(s) \text{ and } \text{cnt}(w(t)) = \text{cnt}(w(s)) \text{ and } \]
\[ \text{set}(w(t)) = \text{set}(w(s)) \text{ and } \]
\[ (\forall j: (j \neq k \text{ and next}(t)(j) = r4) \text{ or } (j \neq k \text{ and next}(t)(j) = next(s)(j))) \text{ and } \]
\[ \text{next}(t)(\text{PID}(t)) \rightarrow EDP \]

\[ \text{ip15}(a) \]

\[ \text{ip16}(a) \]

\[ \text{ip17}(a) | \text{rdcnt}(t) = \text{rdcnt}(a) \text{ and } \text{rd}(t) = \text{rd}(a) \text{ and } \]
\[ \text{wt}(t) = \text{wt}(s) \text{ and } \text{cnt}(w(t)) = \text{cnt}(w(s)) \text{ and } \]
\[ \text{set}(w(t)) = \text{set}(w(s)) \text{ and } \]
\[ (\forall j: (j \neq k \text{ and next}(t)(j) = r5) \text{ or } (j \neq k \text{ and next}(t)(j) = next(s)(j))) \text{ and } \]
\[ \text{next}(t)(\text{PID}(t)) \rightarrow EDP \]
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[ \text{\ldots} \]

\[
\begin{align*}
\text{(next}(s)(j) &= \text{waitAtPm1 and next}(t)(j) = \text{riseAtPm1}) \lor \\
\text{(next}(s)(j) &= \text{waitAtPm2 and next}(t)(j) = \text{riseAtPm2}))) \\
\text{or} \quad (j = k \text{ and } \text{not difference(set}(a)(s), \text{set}(a)(t))(j)) \lor \\
\text{(next}(s)(j) /\text{waitAtPm1 and next}(s)(j) /\text{waitAtPm2})) \\
\text{and next}(t)(j) = \text{next}(s)(j)) \text{ and} \\
\text{next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp18(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp19(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp20(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp21(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp22(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp23(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp24(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp25(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]

\[
\begin{align*}
\text{lp26(s)}: \text{rdcnt}(t) = \text{rdcnt}(s) \text{ and rd}(t) = \text{rd}(s) \text{ and} \\
\text{vt}(t) &= \text{vt}(a) \text{ and cnt}(a(t)) = \text{cnt}(a(s)) \text{ and} \\
\text{set}(a(t)) &= \text{set}(a(s)) \text{ and cnt}(v(t)) = \text{cnt}(v(s)) \text{ and} \\
\text{set}(v(t)) &= \text{set}(v(s)) \text{ and} \\
(\forall j: (j = k \text{ and next}(t)(j) = r7) \text{ or} \\
(j = k \text{ and next}(t)(j) = \text{waitAtPm2}))) \\
\text{or} \quad (j = k \text{ and next}(t)(j) = \text{waitAtPm1})) \\
\text{and next}(t)(\text{pid}(t)) /\text{EOP} \\
\end{align*}
\]
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[
v_t(t) = v_t(s) and cnt(a(t)) = cnt(a(s)) and \%
cnt(w(t)) = cnt(w(s)) + 1 and set(a(t)) = set(a(s)) and X
\]
\[
\text{assert (p: index): (set(v)(p)) and X}
\]
\[
\text{set}(v(t)) = \text{remove}(p, \text{set}(v(s))))) and X
\]
\[
\text{for all } j: (j /\not= k \text{ and next}(t)(k) = r(t)) or X
\]
\[
(j /\not= k \text{ and } \text{difference}(set(v(s)), set(v(t)))) and X
\]
\[
\text{and next}(a(t)) = \text{waitAtPw} and next(t)(j) = \text{riseAtPw}) X
\]
\[
or (j /\not= k \text{ and } \text{(NOT difference}(set(v(s)), set(v(t)))) and X
\]
\[
or \text{next}(a(t)) = \text{waitAtPw} and X
\]
\[
\text{next}(t)(j) = \text{next}(a(t))) and next(t)(pID(t)) /\not= \text{EXP}) X
\]
\[
\text{---------------------------------------------------------------------------------------------------------------------}
\]
\[
lp28(s)
\]
\[
-----------------------------------------------
\]
\[
lp29(a)
\]
\[
-----------------------------------------------
\]
\[
lp29(s)
\]
\[
-----------------------------------------------
\]
\[
lp30(s)
\]
\[
-----------------------------------------------
\]
\[
lp31(s)
\]
\[
-----------------------------------------------
\]
\[
lp32(a)
\]
\[
-----------------------------------------------
\]
\[
lp32(s)
\]
\[
-----------------------------------------------
\]
\[
lp33(s)
\]
\[
-----------------------------------------------
\]
\[
lp34(s)
\]
\[
-----------------------------------------------
\]
\[
lp36(a)
\]
\[
-----------------------------------------------
\]
\[
lp36(s)
\]
\[
-----------------------------------------------
\]
\[
lp38(s)
\]
\[
-----------------------------------------------
\]
\[
lp39(s)
\]
\[
-----------------------------------------------
\]
(forall j: next(t)(j) = next(a)(j)) and
next(t)(pid(t)) /\ EDP

forall (1: inclex): next(t)(i) = EDP)

(getinv.pvs)

getinv: THEORY
BEGIN
importing transition
s: VAN state
p: VAN state
1: VAN 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(w(t)) = 1 and empty?(set(w(t))) and
cnt(w(t)) = 1 and empty?(set(w(t))) and
rd(t) = 0 and wt(t) = 0 and rdct(t) = 0 and
forall i: (i < n and next(t)(i) = ri) or (i > n and next(t)(i) = wt)

forall t: (initcond(t) \implies rp(t))
and (forall s, t: rp(s) and trans(s, t) \implies rp(t))

% after the first iteration from failed proofs of crpII
% we read the invariants as below
C. The Readers/Writers Model in SPIN, SAL, and PVS

\begin{verbatim}
S7(t): bool = rdct(t) >= 0
S2(t): bool = cnt(w(t)) <= 1
S1(t): bool = cnt(a(t)) <= 1
S6(t): bool = cnt(w(t)) = 1 \Rightarrow (wt(t) = 0 and rd(t) = 0)
S91(t): bool = forall i: next(t)(i) = riaseAtPwr
implies wt(t) = 0
S31(t): bool = forall 1:
    next(t)(1) = r4
implies cnt(m(t)) <= 0
S32(t): bool = forall 1:
    next(t)(1) = r9
implies cnt(m(t)) <= 0
S41(t): bool = forall 1: next(t)(1) = r3
implies rdct(t) = rd(t) + 1
S6(t): bool = forall 1: next(t)(1) = r8
implies rd(t) = rdct(t) + 1
S81(t): bool = forall 1: next(t)(1) = riaseAtPwr
implies wt(t) = 0
S101(t): bool = forall 1: next(t)(i) = w3
implies wt(t) = 1

ind1(t): bool = rp(t) and S1(t) and S2(t) and S3(t) and S6(t) and S7(t) and S81(t) and S91(t) and S101(t)
implies wt(t) = 1

\end{verbatim}

% from the unprovable sequent we got more

\begin{verbatim}
X\text{invariants, which together with previous ones, give us new set:}
S92(t): bool = forall 1: next(t)(i) = riaseAtPwr
implies cnt(w(t)) <= 0
S36(t): bool = forall 1:
    next(t)(1) = r3 implies cnt(a(t)) <= 0
S38(t): bool = forall 1: next(t)(1) = r8 implies cnt(a(t)) <= 0
S9(t): bool = forall 1:
    next(t)(1) = riaseAtPwr implies cnt(a(t)) <= 0
S10(t): bool = forall 1: next(t)(1) = w2
implies wt(t) = 1
S83(t): bool = forall 1: next(t)(1) = riaseAtPwr
implies cnt(w(t)) <= 0
S11(t): bool = bool = forall 1: next(t)(1) = r7
implies rd(t) = rdct(t)
S12(t): bool = bool = forall 1: next(t)(1) = r2
implies rd(t) = rdct(t)
S126(t): bool = bool = forall 1: next(t)(1) = r2
implies rd(t) >= 1
ind2(t): bool = ind1(t) and S92(t) and S36(t) and S92(t) and S38(t) and S9(t) and S10(t) and S83(t) and S11(t) and S12(t) and S126(t)

\end{verbatim}

\begin{verbatim}
X\text{new invariants:}
S34(t): bool = forall i: next(t)(i) = r2
implies cnt(a(t)) <= 0
S37(t): bool = bool = forall i: next(t)(i) = r7
implies cnt(a(t)) <= 0
S114(t): bool = bool = forall i: next(t)(i) = riaseAtPwr
implies rd(t) = rdct(t)
S115(t): bool = bool = forall i: next(t)(i) = riaseAtPwr
implies rd(t) = rdct(t)
\end{verbatim}
S131(t): \text{bool} = \forall i: \text{next}(t)(i) = r1 \land \text{cnt}(a(t)) = 1 \\
\implies \text{rd}(t) = \text{rdcnt}(t)

S123(t): \text{bool} = \forall i: \text{next}(t)(i) = r6 \\
\implies \text{rd}(t) > 1

S124(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r1setPw2} \\
\implies \text{rd}(t) > 1

S133(t): \text{bool} = \forall i: \text{next}(t)(i) = r6 \land \text{cnt}(m(t)) = 1 \\
\implies \text{rd}(t) > \text{rdcnt}(t)

\text{ind3}(t): \text{bool} = \text{ind2}(t) \land S34(t) \land S37(t) \land S114(t) \\
\land \left( \text{S115(t)} \land S123(t) \land S124(t) \land S131(t) \land S133(t) \right)

\text{crpin3}: \text{lemma} (\forall t: \text{in1tcond}(t) \Rightarrow \text{ind3}(t)) \\
\text{and} \forall s, t: (\text{ind3}(s) \land \text{trans}(a, t) \Rightarrow \text{ind3}(t)) \chi 6000

S33(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r1setPw1} \\
\implies \text{cnt}(a(t)) < 0

S36(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r1setPw2} \\
\implies \text{cnt}(a(t)) < 0

S122(t): \text{bool} = \forall i: \text{next}(t)(i) = r5 \\
\implies \text{rd}(t) > 1

S132(t): \text{bool} = \forall i: \text{next}(t)(i) = r6 \land \text{cnt}(a(t)) = 1 \\
\implies \text{rd}(t) = \text{rdcnt}(t)

\text{ind4}(t): \text{bool} = \text{ind3}(t) \land S33(t) \land S36(t) \land S122(t) \land S132(t)

\text{crpin4}: \text{lemma} (\forall t: \text{in1tcond}(t) \Rightarrow \text{ind4}(t)) \\
\text{and} \forall s, t: (\text{ind4}(s) \land \text{trans}(a, t) \Rightarrow \text{ind4}(t)) \chi 5620

S113(t): \text{bool} = \forall i: \text{next}(t)(i) = r4 \\
\implies \text{rd}(t) = \text{rdcnt}(t)

S121(t): \text{bool} = \forall i: \text{next}(t)(i) = r4 \\
\implies \text{rd}(t) > 1

\text{ind5}(t): \text{bool} = \text{ind4}(t) \land S113(t) \land S121(t)

\text{crpin5}: \text{lemma} (\forall t: \text{in1tcond}(t) \Rightarrow \text{ind5}(t)) \\
\text{and} \forall s, t: (\text{ind5}(s) \land \text{trans}(a, t) \Rightarrow \text{ind5}(t)) \chi 5737

S42(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r1setPwPer} \\
\implies \text{rdcnt}(t) = \text{rd}(t) + 1

\text{ind6}(t): \text{bool} = \text{ind5}(t) \land S42(t)

\text{crpin6}: \text{lemma} (\forall t: \text{in1tcond}(t) \Rightarrow \text{ind6}(t)) \\
\text{and} \forall s, t: (\text{ind6}(s) \land \text{trans}(a, t) \Rightarrow \text{ind6}(t)) \chi 273

\text{END getinv}

5. invj.pvs

invj X [ parameters ]
: THEORY
% assuming declarations
% EDASSUMING

BEGIN

% assuming declarations
% EDASSUMING
importing transition
m: VAR stateup
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[ t: \text{VAR state} \]

\[ i, j: \text{VAR index} \]

\[ \text{Safety property:} \]

\[ \text{rp}(t): \text{bool} = (vt(t) = 0 \text{ or } rd(t) = 0) \text{ and } vt(t) < 2 \text{ and } rd(t) > 0 \]

\[ \text{and } vt(t) > 0 \]

\[ \text{Initial state:} \]

\[ \text{initcond}(t): \text{bool} = \text{cnt}(m(t)) = 1 \text{ and empty?(set(m(t))) and} \]

\[ \text{cnt}(w(t)) = 1 \text{ and empty?(set(w(t))) and} \]

\[ rd(t) = 0 \text{ and vt(t) = 0 and } \text{rdcnt}(t) = 0 \text{ and} \]

\[ \text{(forall } i: \text{index): } (i < a \text{ and } \text{next}(t)(i) = r1) \]

\[ \text{or } (i > a \text{ and } \text{next}(t)(i) = w1)) \]

\[ \text{Invariants as found in } \text{inlcong.pvs} \]

\[ \text{51}(t): \text{bool} = (\text{cnt}(a(t)) < a) \text{ and} \]

\[ \text{S2}(t): \text{bool} = (\text{cnt}(v(t)) < 1) \]

\[ \text{S6}(t): \text{bool} = (\text{cnt}(v(t)) > 1) \text{ and } (vt(t) = 0 \text{ and } rd(t) = 0) \]

\[ \text{S7}(t): \text{bool} = (\text{rdcnt}(t) > 0) \]

\[ \text{S31}(t): \text{bool} = (\text{next}(t)(i) = r4) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S32}(t): \text{bool} = (\text{next}(t)(i) = r9) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S33}(t): \text{bool} = (\text{next}(t)(i) = r1) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S34}(t): \text{bool} = (\text{next}(t)(i) = r2) \text{ implies } \text{cnt}(m(t)) <= 0 \text{ done, 3638.12} \]

\[ \text{S36}(t): \text{bool} = (\text{next}(t)(i) = r3) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S36}(t): \text{bool} = (\text{next}(t)(i) = r3) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S38}(t): \text{bool} = (\text{next}(t)(i) = r8) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S39}(t): \text{bool} = (\text{next}(t)(i) = r9) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S41}(t): \text{bool} = (\text{next}(t)(i) = r3) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S42}(t): \text{bool} = (\text{next}(t)(i) = r4) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S5}(t): \text{bool} = (\text{next}(t)(i) = r7) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S38}(t): \text{bool} = (\text{next}(t)(i) = r8) \text{ implies } \text{cnt}(m(t)) <= 0 \]

\[ \text{S41}(t): \text{bool} = (\text{next}(t)(i) = r3) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S42}(t): \text{bool} = (\text{next}(t)(i) = r4) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S5}(t): \text{bool} = (\text{next}(t)(i) = r7) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S81}(t): \text{bool} = (\text{next}(t)(i) = r1) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S82}(t): \text{bool} = (\text{next}(t)(i) = r2) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S83}(t): \text{bool} = (\text{next}(t)(i) = r3) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S84}(t): \text{bool} = (\text{next}(t)(i) = r4) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S85}(t): \text{bool} = (\text{next}(t)(i) = r5) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S86}(t): \text{bool} = (\text{next}(t)(i) = r6) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S87}(t): \text{bool} = (\text{next}(t)(i) = r7) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S88}(t): \text{bool} = (\text{next}(t)(i) = r8) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S89}(t): \text{bool} = (\text{next}(t)(i) = r9) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S90}(t): \text{bool} = (\text{next}(t)(i) = r1) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S91}(t): \text{bool} = (\text{next}(t)(i) = r2) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S92}(t): \text{bool} = (\text{next}(t)(i) = r3) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S93}(t): \text{bool} = (\text{next}(t)(i) = r4) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S94}(t): \text{bool} = (\text{next}(t)(i) = r5) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S95}(t): \text{bool} = (\text{next}(t)(i) = r6) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S96}(t): \text{bool} = (\text{next}(t)(i) = r7) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S97}(t): \text{bool} = (\text{next}(t)(i) = r8) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]

\[ \text{S98}(t): \text{bool} = (\text{next}(t)(i) = r9) \text{ implies } \text{rdcnt}(t) = rd(t) + 1 \]
S114(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{rlsAtPwr}
\implies \text{rd}(t) = \text{rdcnt}(t)
\text{s115(t): bool} = \forall i: \text{next}(t)(i) = \text{rlsAtPwr}
\implies \text{rd}(t) = \text{rdcnt}(t)
S121(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r4}
\implies \text{rd}(t) = 1
S122(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r6}
\implies \text{rd}(t) = 1
S123(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r6}
\implies \text{rd}(t) = 1
S124(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{rlsAtPwr}
\implies \text{rd}(t) = 1
S125(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r7}
\implies \text{rd}(t) = 1
S131(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r1}
\text{and cnt}(a(t)) = 1
\implies \text{rdcnt}(t) = \text{rd}(t)
S132(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r6}
\text{and cnt}(a(t)) = 1
\implies \text{rdcnt}(t) = \text{rd}(t)
S133(t): \text{bool} = \forall i: \text{next}(t)(i) = \text{r6}
\text{and cnt}(a(t)) = 1
\implies \text{rdcnt}(t) = \text{rd}(t)
\text{CS1p(t, j): bool} = \forall i: \text{next}(t)(i) = \text{rlsAtPwr2}
\text{or} \text{next}(t)(i) = \text{r2}
\text{or} \text{next}(t)(i) = \text{r3}
\text{or} \text{next}(t)(i) = \text{r4}
\text{or} \text{next}(t)(i) = \text{rlsAtPwr2}
\text{or} \text{next}(t)(i) = \text{r7}
\text{or} \text{next}(t)(i) = \text{r8}
\text{or} \text{next}(t)(i) = \text{r9}
\text{or} \text{next}(t)(i) = \text{waitAtPwr}
\text{or} \text{next}(t)(i) = \text{rlsAtPwr}
\text{CS2(t): bool} = \forall i: \text{next}(t)(i) = \text{v2}
\text{or} \text{next}(t)(i) = \text{v3}
\text{or} \text{next}(t)(i) = \text{rlsAtPwr2}
\text{or} \text{next}(t)(i) = \text{rlsAtPwr}
\text{CS3(t): bool} = \forall i: \text{next}(t)(i) = \text{v2}
\text{or} \text{next}(t)(i) = \text{v3}
\text{or} \text{next}(t)(i) = \text{rlsAtPwr2}
\text{or} \text{next}(t)(i) = \text{rlsAtPwr}
\text{indc(t): bool} = \text{CS1(t)} \text{and} \text{CS2(t)} \text{and} \text{rp}(t) \text{and} \text{S1(t)} \text{and} \text{S2(t)}
\text{and} \text{S31(t)} \text{and} \text{S32(t)} \text{and} \text{S33(t)} \text{and} \text{S34(t)} \text{and} \text{S35(t)}
\text{and} \text{S36(t)} \text{and} \text{S37(t)} \text{and} \text{S38(t)} \text{and} \text{S39(t)}
\text{and} \text{S41(t)} \text{and} \text{S42(t)} \text{and} \text{S43(t)} \text{and} \text{S46(t)}
\text{and} \text{S7(t)} \text{and} \text{S81(t)} \text{and} \text{S82(t)} \text{and} \text{S83(t)}
\text{and} \text{S91(t)} \text{and} \text{S92(t)} \text{and} \text{S10(t)} \text{and} \text{S101(t)}
\text{and} \text{S111(t)} \text{and} \text{S112(t)} \text{and} \text{S113(t)} \text{and} \text{S114(t)} \text{and} \text{S115(t)}
\text{and} \text{S121(t)} \text{and} \text{S122(t)} \text{and} \text{S123(t)} \text{and} \text{S124(t)} \text{and} \text{S125(t)}
\text{and} \text{S131(t)} \text{and} \text{S132(t)} \text{and} \text{S133(t)}
\text{crrndrp: lemma} \forall \text{t: initcond(t) => rp(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => rp(t)) S1464, 1851xipd}
\text{crrn1: lemma} \forall \text{t: initcond(t) => S1(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => S1(t)) S160-30(s1)}
\text{crrn2: lemma} \forall \text{t: initcond(t) => S2(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => S2(t)) S30-218(s1)}
\text{crrn3: lemma} \forall \text{t: initcond(t) => S3(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => S3(t)) S30-4670}
\text{crrn4: lemma} \forall \text{t: initcond(t) => S7(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => S7(t)) S130(s1)}
\text{crrn5: lemma} \forall \text{t: initcond(t) => S81(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => S81(t)) S146440(s1)}
\text{crrn6: lemma} \forall \text{t: initcond(t) => S33(t)}
\text{and forall s, t: (indc(s)}
\text{and trans(s, t) => S33(t)) Snew-694(s2+s1)}
C. The Readers/Writers Model in SPIN, SAL, and PVS

and forall s, t: (indc(s) and trans(s, t) \ involv-286(s"r")

crped34: lemma (forall t: initcond(t) \ involv-298(s"r")
and forall s, t: (indc(s) and trans(s, t) \ involv-354(s"r")

crped35: lemma (forall t: initcond(t) \ involv-403
and forall s, t: (indc(s) and trans(s, t) \ involv-471(s"r")

crped36: lemma (forall t: initcond(t) \ involv-536(s"r")
and forall s, t: (indc(s) and trans(s, t) \ involv-644(s"r")

\ we found another invariant while proving crped39:
S140(t): bool \ forall i: next(t)(i) = waitAtPwr
implies cmax(t(u)) <= 0

indci(t): bool \ forall s, t: (indc(s) and S140(t)

and forall s, t: (indc(s) and S140(t)

\ %"divide and conquer" CS1 and CS2, so that proof would be faster
CS1(t): bool =
forall i, j: next(t)(i) \ releastPw1
and CS1pred(t, j)
\ 1 = j

CS12(t): bool =
forall i, j: next(t)(i) \ r2
and CS1pred(t, j)
\ 1 = j

CS13(t): bool =
forall i, j: next(t)(i) \ r3
and CS1pred(t, j)
\ 1 = j

CS14(t): bool =
forall i, j: next(t)(i) \ r4
and CS1pred(t, j)
\ 1 = j

CS15(t): bool =
forall i, j: next(t)(i) \ releastPw2
and CS1pred(t, j)
\ 1 = j

CS16(t): bool =
forall i, j: next(t)(i) \ r7
and CS1pred(t, j)
\ 1 = j

CS17(t): bool =
forall i, j: next(t)(i) \ r6
and CS1pred(t, j)
\ 1 = j

CS18(t): bool =
forall i, j: next(t)(i) \ r9
and CS1pred(t, j)
\ 1 = j

CS19(t): bool =
forall i, j: next(t)(i) \ waitAtPwr
and CS1pred(t, j)
\ 1 = j

CS110(t): bool =

\
forall 1, j: \text{next}(t)(i) = \text{isAtPwr}
and \text{CS2prod}(t, j) \\
\Rightarrow i = j

\text{CS21}(t): \text{bool} =
forall 1, j: \text{next}(t)(i) = w2 
and \text{CS2pred}(t, j) \\
\Rightarrow i = j

\text{CS22}(t): \text{bool} =
forall 1, j: \text{next}(t)(i) = w3 
and \text{CS2prod}(t, j) \\
\Rightarrow i = j

\text{CS23}(t): \text{bool} =
forall 1, j: \text{next}(t)(i) = \text{isAtPwr} 
and \text{CS2pred}(t, j) \\
\Rightarrow i = j

\text{CS24}(t): \text{bool} =
forall 1, j: \text{next}(t)(i) = \text{isAtPwr} 
and \text{CS2prod}(t, j) \\
\Rightarrow i = j

\text{indcs11}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS11}(t)) \text{#new=3443}

\text{indcs12}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t)) \text{#new=4026}
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS12}(t))

\text{indcs13}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) \text{#new=3277} 
and \text{trans}(s, t) \Rightarrow \text{CS13}(t))

\text{indcs14}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS14}(t)) \text{#new=3252}

\text{indcs15}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) \text{#new=2159} 
and \text{trans}(s, t) \Rightarrow \text{CS15}(t))

\text{indcs16}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS16}(t)) \text{#new=3198}

\text{indcs17}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS17}(t)) \text{#new=3198}

\text{indcs18}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS18}(t)) \text{#new=3373}

\text{indcs19}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS19}(t)) \text{#new=3444}

\text{indcs20}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS20}(t)) \text{#new=3505}

\text{indcs21}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS21(t)}) \text{#new=3137}

\text{indcs22}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS22}(t)) \text{#new=3084}

\text{indcs23}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS23(t)}) \text{#new=2940}

\text{indcs24}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS24(t)}) \text{#new=2794}

\text{indcs25}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{CS1}(t))
and \forall s, t: (\text{indcl}(s) 
and \text{trans}(s, t) \Rightarrow \text{CS25(t)}) \text{#new=17}

\text{crpind41}: \text{lemma} (\forall t: \text{initcond}(t) \Rightarrow \text{S41(t)})
C. The Readers/Writers Model in SPIN, SAL, and PVS

and forall s, t: (indc1(s)
and trans(s, t) \rightarrow S41(t) Xnew-663(s,r*)

crplnd42: lemma (forall t: initcond(t) \rightarrow S42(t))
and forall s, t: (indc1(s)
and trans(s, t) \rightarrow S42(t) Xnew-705

% This is where we need S43
S43(t): bool = forall i: next(t)(i) = waitAtPow
implies rdcont(t) = rd(t) + 1
indc2(t): bool = indc1(t) and S43(t)
crplnd43: lemma (forall t: initcond(t) \rightarrow S43(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S43(t) Xnew-664(s,r*)
crplnd44: lemma (forall t: initcond(t) \rightarrow S44(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S44(t) Xnew-726(s,r* and revins)
crplnd5: lemma (forall t: initcond(t) \rightarrow S41(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S6(t) Xnew-690(s,r*)
crplnd10: lemma (forall t: initcond(t) \rightarrow S10(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S10(t) Xnew-546(s,w*)
crplnd101: lemma (forall t: initcond(t) \rightarrow S101(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S101(t) Xnew-603(s,w*)
crplnd81: lemma (forall t: initcond(t) \rightarrow S81(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S81(t) Xnew-617,but
I've had to use S81 (indc1: s "w")
crplnd82: lemma (forall t: initcond(t) \rightarrow S82(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S82(t) Xnew-567(s,w*+s83revins)
crplnd63: lemma (forall t: initcond(t) \rightarrow S63(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S63(t) Xnew-406(s,w*+rev)
crplnd91: lemma (forall t: initcond(t) \rightarrow S91(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S91(t) Xnew-612(s,w*) rev(S92)
crplnd92: lemma (forall t: initcond(t) \rightarrow S92(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S92(t) Xnew-650 too,
Xadd after all for one goal and inst-730

END invj

6. invjl.pvs

invjl: THEORY

BEGIN

importing invj
s: VAR state
x: VAR state

crplnd11: lemma (forall t: initcond(t) \rightarrow S11(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S11(t) Xnew-684

crplnd12: lemma (forall t: initcond(t) \rightarrow S12(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S12(t) Xnew-456

crplnd13: lemma (forall t: initcond(t) \rightarrow S13(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S13(t) Xnew-526

crplnd14: lemma (forall t: initcond(t) \rightarrow S14(t))
and forall s, t: (indc2(s)
and trans(s, t) \rightarrow S14(t) Xnew-604 we need another:
S150(t): bool = forall i: (index): next(t)(i) = r9
implies rdcont(t) = rd(t)
C. The Readers/Writers Model in SPIN, SAL, and PVS

\begin{verbatim}

indc4(t): bool = indc3(t) and S160(t)
crpind141: lemma (forall t: initcond(t) \implies S14(t))
    and forall s, t: (indc3(s)
    and trans(s, t) \implies S14(t))% new 792

crpind150: lemma (forall t: initcond(t) \implies S150(t))
    and forall s, t: (indc3(s)
    and trans(s, t) \implies S150(t))% new-1477

crpind115: lemma (forall t: initcond(t) \implies S115(t))
    and forall s, t: (indc3(s)
    and trans(s, t) \implies S115(t))% new-786

crpind121: lemma (forall t: initcond(t) \implies S121(t))
    and forall s, t: (indc3(s)
    and trans(s, t) \implies S121(t))% new-792

crpind122: lemma (forall t: initcond(t) \implies S122(t))% new for those 3
    and forall s, t: (indc3(s)
    and trans(s, t) \implies S122(t))% new-1160

END invjl

7. cardsem.pvs

    cardsem : THEORY

    BEGIN
    importing invjl

    s: VAR stateexp
    t: VAR state

    i: VAR index

    P(t, i): bool = next(t)(i) = r3 or next(t)(i) = r4
        or next(t)(i) = r6 or next(t)(i) = r6
        or next(t)(i) = r7 or next(t)(i) = waitAtPwr
        or next(t)(i) = raiseAtPwr or next(t)(i) = waitAtPwr
        or next(t)(i) = raiseAtPwr

    aux: finite_set[index] = {i: index | P(t, i)}

    aux: lemma forall (t: state): (exists (i: index): next(t)(i) = r4
        or next(t)(i) = r6 or next(t)(i) = r6
        or next(t)(i) = r7 or next(t)(i) = waitAtPwr
        or next(t)(i) = raiseAtPwr or next(t)(i) = waitAtPwr
        or next(t)(i) = raiseAtPwr) \implies card(aux(t)) \geq 1

    a(t): bool = card(aux(t)) \geq rdcnt(t)

    indc4(t): bool = indc3(t) and a(t)

    a_inv: lemma (forall t: initcond(t) \implies a(t))
        and forall s, t: (indc4(s)
        and trans(s, t) \implies a(t))% new-348

    crpind1221: lemma (forall t: initcond(t) \implies S122(t))
        and forall s, t: (indc4(s)
        and trans(s, t) \implies S122(t))% new-130

END invjl
\end{verbatim}
C. The Readers/Writers Model in SPIN, SAL, and PVS

8. dq.pvs

dq: THEORY

BEGIN
importing cardaem
s: VAR state
i, t, u: VAR state
DQdecrease(s, t): bool = (exists i: IntRW(next(a)(i)) > IntP.W(next(t)(i))) and (forall i: IntP.W(next(a)(i)) > IntRW(next(t)(i)))
dq: theorem (forall s: (indc4(a) •> S123(t))) and (forall s, t: (indc4(a) •> S124(t))) and
    (forall i: next(a)(i) • next(t)(i)) and
    rd(s) = rd(t) and wt(s) = wt(t) •> DQdecrease(s, t)) •> S125(t)
dqb: lemma (forall s1: (indc4(a1) •> (forall i: IntRW(next(s1)(i)) = 0)) or (exists i: IntRW(next(s1)(i)) > 0)) and
    (forall i: IntRW(next(s1)(i)) = 0) and
    (exists i: IntP.W(next(s1)(i)) > 0) and
    (forall i: IntP.W(next(s1)(i)) = 0) and
    (exists i: IntP.W(next(s1)(i)) > 0)
    dqbinv1(t): bool = (forall i: IntRW(next(t)(i)) = 0) or (exists i: IntRW(next(t)(i)) > 0)
    dqbinv2(t): bool = (forall i: IntP.W(next(t)(i)) = 0) or (exists i: IntP.W(next(t)(i)) > 0)
    dqbinv3(t): bool = (forall i: IntRPP(next(t)(i)) = 0) or (exists i: IntRPP(next(t)(i)) > 0)

The proofs of the three previous theorems pretty much alike.

END
next(t) = r7 or next(t) = r8 or next(t) = r9 or
(next(t) = r10 and cnt(t) < 0) or
(next(t) = w2 and cnt(w(t)) < 0) or
(next(t) = w3 and cnt(w(t)) < 0)
dqb21: lemma \( \forall t: \text{initcond}(t) \Rightarrow \text{dqpiv2}(t) \)
and \( \forall s, t: (\text{dpbiv2}(s) \land \text{ind}(s)) \)
and \( \text{trans}(s, t) \Rightarrow \text{dqpiv2}(t) \)
\( \exists \text{set}(t): \text{bool} = \text{cnt}(\text{m}(t)) < 0 \Rightarrow \exists ! [\text{set}(m(t))]) \)
\( \exists \text{setm}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \forall t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{set}(m(t)) \\
\exists \text{setv2}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setv1}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc1}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setcl}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc2}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setcl}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw} \\
\exists \text{setc3}(t): \text{bool} = \forall t: \text{set}(s(t)) \Rightarrow \text{next}(t) = \text{waitAtPw} \\
\text{or} \exists t: \text{set}(s(t)) \Rightarrow \text{waitAtPw}
C. The Readers/Writers Model in SPIN, SAL, and PVS

dqbinv2: lemma (forall t: initcond(t) => dqbinv2(t))
and forall s, t: (indc(s)
and trans(s, t) => dqbinv2(t)) Unprovable, we need:
P(t, t): bool = next(t)(i) = r4
or next(t)(i) = r6 or next(t)(i) = r6
or next(t)(i) = r7 or next(t)(i) = r8
or next(t)(i) = waitAtPw2
or next(t)(i) = releasePw2
cr(t): finite_set[index] = {i: index | P(t, i)}
craux: lemma forall (t: state): (exists (i: index): next(t)(i) = r4
or next(t)(i) = r6 or next(t)(i) = r6
or next(t)(i) = r7 or next(t)(i) = r8
or next(t)(i) = waitAtPw2
or next(t)(i) = releasePw2) => card(cr(t)) >= 1
cr1(t) :bool = card(cr(t)) = rd(t)
indc6(t): bool = indc6(t) and cr1(t)
cri(t): lemma (forall t: initcond(t) => cri(t))
and forall s, t: (indc(s)
and trans(s, t) => cri(t)) Xnew 569
dqbinv2final: lemma (forall t: initcond(t) => dqbinv2(t))
and forall s, t: (indc(s)
and trans(s, t) => dqbinv2(t)) Xnew1777
dqbinv3: lemma (forall t: initcond(t) => dqbinv3(t))
and forall s, t: (indc(s)
and trans(s, t) => dqbinv3(t)) XFor the last goal
Xwe need dqinv4
dqinv4(t): bool = forall i: (next(t)(i) = releasePw or next(t)(i) = x2
or next(t)(i) = x3 and cnt(t)(i) <= 0 and cnt(t)(i) <= 0)
and (forall (k: index): next(t)(k) /= waitAtPw)
=> exists (k: index):
next(t)(k) = releasePw or next(t)(k) = r2 or
next(t)(k) = r3 or next(t)(k) = releasePw or
next(t)(k) = r4 or next(t)(k) = releasePw2 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) => dqinv4(t))
and forall s, t: (indc7(s)
and trans(s, t) => dqinv4(t)) Xnew -756
dqinv4: lemma (forall t: initcond(t) => dqinv4(t))
and forall s, t: (indc7(s)
and trans(s, t) => dqinv4(t)) Xnew -2694, with indc7
END dqb

10. dqbfinal.pvs

dqbfinal % [ parameters ]
: THEORY
BEGIN
% ASSUMING
% assuming declarations
% END ASSUMING
IMPORTING dqb
s: VAR state
t0, t: VAR state
i: VAR index
% in order to prove TCC for dqb
dqbinv6(t): bool = next(t)(piD(t)) /= EOP or forall i: next(t)(i) = EOP
indc8(t): bool = indc7(t) and dqbinv6(t)
dqbinv6final: lemma (forall t: initcond(t) => dqinv6(t))
and forall s, t: (indc8(s)
and trans(s, t) => dqinv6(t))
dqbfassisti: lemma nonempty?() {i: index |
C. The Readers/Writers Model in SPIN, SAL, and PVS

next(t)(i) = waitAtPat
AND next(t)(i) = waitAtPw
AND next(t)(i) = waitAtPn
AND next(t)(i) = EOP) and (LET rdcnt = rdcnt(t),
pID =
choose(t: index
next(t)(i) = waitAtPat
next(t)(i) = waitAtPw
next(t)(i) = waitAtPn
AND next(t)(i) = EOP),
rd = rd(t),
vt = vt(t),
cuta = cut(a(t)),
seta = set(t(a)),
cutv = cut(v(t)),
setz = set(v(t)),
next = next(t)
IN (# pID := pID,
m := (# cnt := cuta, set := seta #),
v := (# cnt := cutv, set := setv #),
rdcnt := rdcnt,
next := next,
rd := rd,
vt := vt #)

~ 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 vt(s) = vt(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 vt(s) = vt(t))

Xnsv-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 vt(s) = vt(t)))

X120

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 vt(s) = vt(t)))

X181

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 vt(s) = vt(t)))

X183

dqbassist: lemma forall s: indc8(s) =
(p4(s) or p13(s) or p22(s) or p26(s) =
(exists t: trans(s, t) and exists u: trans(s, 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 vt(t) = vt(u)))

dqb: lemma forall s1: indc8(s1) =
!!!!!!!!!!!

!!!

!!!

!!!

!!!

!!!

!!!

!!!

!!!

!!!

!!!

!!!

!!!
ordering

ordering
: THEORY
BEGIN
importing dqbfinal
s: VAR state
t: VAR state
active(i: label): nat = if i = 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 1
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 1
DQtotal(s, t): bool = table
X---------------------------------X
| Pos(s, M) > Pos(t, M) | TRUE | II
| Pos(s, M) = Pos(t, M) and SUM(s, M) > SUM(t, M) | TRUE | II
X---------------------------------X
| Pos(s, M) = Pos(t, M) and SUM(s, M) < SUM(t, M) | FALSE | II
X---------------------------------X
| Pos(s, M) < Pos(t, M) | FALSE | II
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 - "plD(t! 1)"
    (flatten)
    (ind_inv1)$
    (branch (split +)
      (then
        (try (skolem!)
        (expand "initcond"
        (flatten)
        (inst - "i!1"
        (grind))(grind)))))))

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

```
(defstep bddtrans ()
 (let ((transvar (gather-fvars (a-forms *goal*)
                           '= ' nil
                           '#(lambda (sf)
                               (and (negation? (formula sf)) (branch? (argsl (formula sf)))))))
   (bddsimp transvar)
   
   "..."
   "..."
)
)

Invariants of type forall (1:index): P(x) => v
(defstep s (arg)
 (branch (split)
   (then (skolem!)
   (flatten)
   (expand "initcond")
   (flatten)
   (indent)$
   (skolem!)
   (inst = "i1")
   (grind)
```

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

(then
  (skolem))
(flatten)
(let ((forms (a-forms (current-goal +pa+))))
  (indinv (string (id (operator (args1
    ..
..)
(fonula (car (select-seq aforms -1)))))))
  (then (if (equal indinv "indc3")
    (expand "indc3" "indc2" "indc1" "indc")
    (expand "indc2" "indc1" "indc")
    (if (equal indinv "indc1")
      (expand "indc1" "indc")
      (expand "indc")))
  (flatten)
  (ind_inv1$)
  (skolema!)
  (inst?)
  (flatten)
  (branch
    (case "111-pID(sl1)"
      ((then
        (hide -2 -3)
        (expand "trans")
        (branch (tasimp)
          ((if (equal (get-goalnum +pa+ 30)
            (then (lemmas "trans_TCC2")
              (branch (inst - "pID(sl1)" "s1!" "s1!"
                "trans")
                (branch (split -1)
                  (if (equal (get-goalnum +pa+) 30)
                    (then (grind))
                    (then (reveal -4)
                      (hide-all-but
                        :keep-fnums
                        (-1 -2 -13 -14 -16 -24
                        -26 -35 -27 -28 -29)
                      (grind))))))
          (then (inst - "pID(s11)"
            (grind))))))))
      (then (inst - "piD(s11)"
        (grind))))))))))

(defstep s1
  0
  (branch
    (Split)
    (grind)
    (then
      (skolea!)
      (flatten)
      (typepred "next(s11)"
        (inst - "pID(s11)"
          (let ((forms (a-forms (current-goal +pa+))))
            (indinv (string (id (operator (args1
              ..
..)
            (fonula (car (select-seq aforms -2)))))))))
            (then (if (equal indinv "indc3")
              (expand "indc3")
              (expand "indc2")
              (if (equal indinv "indc1")
                (expand "indc1")
                (expand "indc")
                (if (equal indinv "indc")
                  (expand "indc")
                  (expand "indc")))))
      (flatten)
      (grind))))))))

** **
(defstep st ()
  (branch
    (split)
    (grind)
    (then
      (skolem!)
      (flatten)
      (typepred "next(s11)"
        (inst - "pID(s11)"
          (let ((forms (a-forms (current-goal +pa+))))
            (indinv (string (id (operator (args1
              ..
..)
            (fonula (car (select-seq aforms -2)))))))))
            (then (if (equal indinv "indc3")
              (expand "indc3")
              (expand "indc2")
              (if (equal indinv "indc1")
                (expand "indc1")
                (expand "indc")
                (if (equal indinv "indc")
                  (expand "indc")
                  (expand "indc")))))
      (flatten)
      (grind))))))))

** **
C. The Readers/Writers Model in SPIN, SAL, and PVS

(defstep ind_inv_us ()
  (let ((s-forms (s-forms (current-goal *ps*)))
        (inv-name (string (id (operator
                              (formula (car (select-seq s-forms 2)))))))
    (expand inv-name))
  ""
  )
(defstep ref_induct()
  (let ((s-forms (s-forms (current-goal *ps*)))
        (refStepName (string (id (formula (car s-forms)))))
        (then (expand refStepName)
               (split)))
  ""
  )
(defstep tnsmp ()
  (let ((transvar (car
                 (gather-fnuma
                  (s-fol'IIB •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 ()
  (let ((transvar (car
              (gather-fnuma
               (s-fol'IIB •goal•)
               nil
               (lambda (sf)
                 (and (negation? (formula sf)) (branch? (args1 (formula sf)))))))
             (split))))
  ""
  )
(defstep s_tcc_au()
  (let ((transvar (car
              (gather-fnuma
               (s-fol'IIB •goal•)
               nil
               (lambda (sf)
                 (and (negation? (formula sf)) (branch? (args1 (formula sf)))))))
             (split))))
  ""))
(defstep s_tcc_aux ()
  (let ((transvar (car
              (gather-fnuma
               (s-fol'IIB •goal•)
               nil
               (lambda (sf)
                 (and (negation? (formula sf)) (branch? (args1 (formula sf)))))))
             (split))))
  ""
  )
C. The Readers/Writers Model in SPIN, SAL, and PVS

(gather-funs
  (e-forms goal)
  -> trans
  'l(lambda (sf)
    (and (negation? (formula sf)) (disjunction?
      (arg1 (formula sf))))))
)

(hide-all-but :keep-funs (-3 -4 -6 -7 -15 -17 -18 -19 -26
  transvar)))

"..."

"...

) Semaphore invariants
(assume cas (inv)
  (let ((invname (concatenate 'string inv "pred")))
  (branch
    (split)
    (then (skolem1)
      (flatten)
      (expand inv)
      (flatten)
      (skolem1)
      (inst - "!!!")
      (grind)
    )
    (then
      (skolem1)
      (flatten)
      (expand* "indcl" "indc")
      (flatten)
      (expand inv invname)
      (terminal)
      (skolem1)
      (inst-cp - "!!1" ".!1")
      (flatten)
      (expand" "S11" "S12" "S13" "S14" "S15" "S16" "S17" "S18" "S19"
        "S20" "S21" "S22" "S23" "S24" "S25" "S26" "S27" "S28" "S29"
        "S30" "S31" "S32" "S33" "S34" "S35" "S36" "S37" "S38" "S39"
        "S40" "S41" "S42" "S43" "S44" "S45" "S46" "S47" "S48" "S49"
        "S50" "S51" "S52" "S53" "S54" "S55" "S56" "S57" "S58" "S59"
        "S60" "S61" "S62" "S63" "S64" "S65" "S66" "S67" "S68" "S69"
        "S70" "S71" "S72" "S73" "S74" "S75" "S76" "S77" "S78" "S79"
        "S80" "S81" "S82" "S83" "S84" "S85" "S86" "S87" "S88"
        "S89" "S90" "S91" "S92" "S93" "S94" "S95" "S96" "S97" "S98"
        "S99"
      (branch
        (case "!!1=piD(s!!1)"
          ((then
              (if (equal inv "CSI") (hide -4 -6) (hide -2 -3))
              (repeat* (inst - ".!!1")
                (expand "trans")
                (branch (tastmp)
                  ((if (equal (get-goalnum *ps*) 30)
                    (then (lemma "trans_TCC")
                      (branch (inst - "piD(s!!1)" "!!1" "!!1")
                        ((branch (split -1)(if (equal inv "CSI")
                          (reveal -12 -13 -26 -26 -27 -66 -74 -82 -83 -83)
                            (reveal -11 -12 -13 -25 -26 -27 -66 -74 -82 -83)
                              (grind))))))
                        (then (inst-cp - "piD(s!!1)" (inst - ".!!1") (grind)) (propax)))))
                  (then (hide-all-but :keep-funs -1)
                    (if (equal inv "CSI")
                      (reveal -12 -13 -26 -26 -27 -66 -74 -82 -83 -83)
                      (reveal -11 -12 -13 -25 -26 -27 -66 -74 -82 -83)
                      (grind))))))
                  (then (inst-cp - "piD(s!!1)" (inst - ".!!1") (grind)))
                )))
            )))
          )))
        )))
      )))
    )
  )
)

If (equal inv "CSI") (hide -6) (hide -1))
  (inst-cp - "!!1" "piD(s!!1)"
  (inst - ".!!1" "piD(s!!1)"
  (repeat* (inst - "!!1")
    (expand "trans")
    (branch (tastmp)
      ((if (equal (get-goalnum *ps*) 30)
        (then (lemma "trans_TCC")
         )))
    )
)
C. The Readers/Writers Model in SPIN, SAL, and PVS

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 => (v(t) = 0 and rd(t) = 0))
S91(t): bool = forall (i: index): next(t)(i) = rlesAtPlv
implies v(t) = 0
S92(t): bool = forall (i: index): next(t)(i) = rlesAtPlv
implies cnt(w(t)) <= 0
S33(t): bool = forall (i: index):
next(t)(i) = rlesAtPlv
implies cnt(m(t)) <= 0
S34(t): bool = forall (i: index):
next(t)(i) = rll
implies cnt(m(t)) <= 0
S35(t): bool = forall (i: index):
next(t)(i) = r3
implies ent(a(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) = rlesAtPlv
implies cnt(w(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(w(t)) <= 0
S39(t): bool = forall (i: index):
next(t)(i) = rlesAtPlv
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) = rlesAtPlv
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) = rlesAtPnv
implies v(t) = 0
S82(t): bool = forall (i: index):
next(t)(i) = rlesAtPnv
implies rd(t) = 0
S83(t): bool = forall (i: index):
next(t)(i) = rlesAtPnv
implies cnt(w(t)) < 1
S10(t): bool = forall (i: index):
next(t)(i) = w2
implies v(t) = 1
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[ S101(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = w3 \]
\[ S111(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r7 \]
\[ S112(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r2 \]
\[ S113(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r4 \]
\[ S114(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = \text{rileSetPw2} \]
\[ S115(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = \text{rileSetPw1} \]
\[ S121(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r4 \]
\[ S122(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r6 \]
\[ S123(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r6 \]
\[ S124(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = \text{rileSetPw2} \]
\[ S125(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r7 \]
\[ S131(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r1 \]
\[ \text{and cnt}(a(t)) = 1 \]
\[ \text{implies\ rdcnt}(t) = \text{rd}(t) \]
\[ S132(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r6 \]
\[ \text{and cnt}(a(t)) = 1 \]
\[ \text{implies\ rdcnt}(t) = \text{rd}(t) \]
\[ S133(t) : \text{bool} = \forall (i: \text{index}) : \text{next}(t)(i) = r6 \]
\[ \text{and cnt}(a(t)) = 1 \]
\[ \text{implies\ rdcnt}(t) = \text{rd}(t) \]
\[ S140(t) : \text{bool} = \forall (i: \text{next}(t)(i)) = \text{waitAtPwr} \]
\[ \text{implies\ cnt}(a(t)) = 0 \]
\[ S43(t) : \text{bool} = \forall (i : \text{next}(t)(i)) = \text{waitAtPwr} \]
\[ \text{implies\ rdcnt}(t) = \text{rd}(t) + 1 \]
\[ S150(t) : \text{bool} = \forall (i : \text{next}(t)(i)) = r9 \]
\[ \text{implies\ rdcnt}(t) = \text{rd}(t) \]
\[ a(t) : \text{bool} = \text{card}(a(t)) = \text{rdcnt}(t), \text{where} \]
\[ a(t) = \text{finite_set[index]} = \{1 : \text{index} | P(t, i)\} \]
\[ P(t, i) : \text{bool} = \text{next}(t)(i) = r3 \text{ or next}(t)(i) = r6 \]
\[ \text{or next}(t)(i) = r5 \text{ or next}(t)(i) = r6 \]
\[ \text{or next}(t)(i) = r7 \text{ or next}(t)(i) = \text{waitAtPwr} \]
\[ \text{or next}(t)(i) = \text{rileSetPw2} \]
\[ \text{or next}(t)(i) = \text{waitAtPwr} \]
\[ \text{CS1(t)} : \text{bool} = \]
\[ \forall (i, j) : \text{index} : \text{CS1pred}(t, i) \text{ and} \]
\[ \text{CS1pred}(t, j) \Rightarrow i = j, \text{where} \]
\[ \text{CS1pred}(t, i) : \text{bool} = \text{next}(t)(i) = \text{rileSetPw1} \]
\[ \text{or next}(t)(i) = r2 \text{ or} \]
\[ \text{next}(t)(i) = r3 \text{ or next}(t)(i) = r4 \]
\[ \text{or next}(t)(i) = \text{rileSetPw2} \]
\[ \text{or next}(t)(i) = r7 \text{ or next}(t)(i) = r8 \]
\[ \text{or next}(t)(i) = r9 \text{ or next}(t)(i) = \text{waitAtPwr} \]
\[ \text{or next}(t)(i) = \text{rileSetPw2} \]
\[ \text{CS2(t)} : \text{bool} = \]
\[ \forall (i, j) : \text{index} : \text{CS2pred}(t, i) \text{ and} \]
\[ \text{CS2pred}(t, j) \Rightarrow i = j, \text{where} \]
\[ \text{CS2pred}(t, i) : \text{bool} = \text{next}(t)(i) = w2 \]
\[ \text{or next}(t)(i) = w3 \text{ or} \]
\[ \text{next}(t)(i) = \text{rileSetPw2} \]
\[ \text{or next}(t)(i) = \text{rileSetPw2} \]
\[ \text{The additional invariants needed for the clean completion proof:} \]
\[ \text{SetSet}(t) : \text{bool} = \forall (i : \text{set}(a(t)))(i) \Rightarrow \text{next}(t)(i) = \text{waitAtPwr} \]
\[ \text{or next}(t)(i) = \text{waitAtPwr} \]
\[ \text{SetSet}(t) : \text{bool} = \forall (i : \text{set}(a(t)))(i) \Rightarrow \text{next}(t)(i) = \text{waitAtPwr} \]
\[ \text{or next}(t)(i) = \text{waitAtPwr} \]
\[ \text{SetSet}(t) : \text{bool} = \text{card}(a(t)) = 0 \Rightarrow \text{card}(\text{set}(a(t))) = \text{abs}(\text{cnt}(a(t))) \]
\[ \text{SetSet}(t) : \text{bool} = \text{card}(a(t)) = 1 \Rightarrow \text{card}(\text{set}(a(t))) = 0 \]
\[ \text{SetSet}(t) : \text{bool} = \text{cnt}(v(t)) = 0 \Rightarrow \text{card}(\text{set}(v(t))) = \text{abs}(\text{cnt}(v(t))) \]
C. The Readers/Writers Model in SPIN, SAL, and PVS

Ssetc(t): bool = cnt(v(t)) = 1 & cnt(set(w(t))) = 0
crf(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) = waitAtPw2
or 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) = waitAtPw2.
c(t): finite_set(index) = {i: index | P1(t, i)}
dqb1nvt(t): bool = forall (i: index): next(t)(i) = waitAtPw1 or
next(t)(i) = waitAtPw2 or next(t)(i) = waitAtPw3 or
next(t)(i) = waitAtPw4 or
exists (j: index): (next(t)(j) = waitAtPw1 and
next(t)(j) = waitAtPw2 and next(t)(j) = waitAtPw3 and
next(t)(j) = waitAtPw4 and
next(t)(j) = EOP)
dqb1nsv(t): bool =
cnt(v(t)) <= 0
=> exists 1:
(next(t)(1) = r4 or next(t)(1) = r5 or
next(t)(1) = r6 or
next(t)(1) = r7 or
next(t)(1) = r8 or
next(t)(1) = r9 or
next(t)(1) = r10 or
next(t)(1) = r11 or
next(t)(1) = r12 or
next(t)(1) = r13 or
next(t)(1) = r14 or
next(t)(1) = r15 or
next(t)(1) = r16 or
next(t)(1) = r17 or
next(t)(1) = r18 or
next(t)(1) = r19 or
next(t)(1) = r20)
dqb1nsv(t): bool =
cnt(v(t)) <= 0
=> exists 1:
(next(t)(1) = r4 or next(t)(1) = r5 or
next(t)(1) = r6 or
next(t)(1) = r7 or
next(t)(1) = r8 or
next(t)(1) = r9 or
next(t)(1) = r10 or
next(t)(1) = r11 or
next(t)(1) = r12 or
next(t)(1) = r13 or
next(t)(1) = r14 or
next(t)(1) = r15 or
next(t)(1) = r16 or
next(t)(1) = r17 or
next(t)(1) = r18 or
next(t)(1) = r19 or
next(t)(1) = r20)
dqb1nsv(t): bool =
cnt(v(t)) <= 0
=> exists 1:
(next(t)(1) = r4 or next(t)(1) = r5 or
next(t)(1) = r6 or
next(t)(1) = r7 or
next(t)(1) = r8 or
next(t)(1) = r9 or
next(t)(1) = r10 or
next(t)(1) = r11 or
next(t)(1) = r12 or
next(t)(1) = r13 or
next(t)(1) = r14 or
next(t)(1) = r15 or
next(t)(1) = r16 or
next(t)(1) = r17 or
next(t)(1) = r18 or
next(t)(1) = r19 or
next(t)(1) = r20)
dqb1nsv(t): bool =
next(t)(pID(t)) = EOP or forall 1: next(t)(i) = EOP

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

rpl(t): bool = vt(t) = 0 or rd(t) = 0
rp2(t): bool = vt(t) < 2
V1(t): bool = rd(t) >= 0
V2(t): bool = vt(t) >= 0
V3(t): bool = (rdcnt(t) >= 0)
V4(t): bool = (cnt(v(t)) <= 1)
V5(t): bool = (cnt(m(t)) <= 1)
V6(t): bool = (cnt(w(t)) <= 1 & (vt(t) = 0 and rd(t) = 0))
V7(t): bool = (rdcnt(t) >= 1 & rd(t) = 1)
V9(t): bool = (cnt(w(t)) < 1 & ((vt(t) = 1 and rd(t) = 0) or
(rd(t) >= 1 and vt(t) = 0) or
(rd(t) = 0 and wt(t) = 0) and
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) = r9 or
next(t)(i) = r10 or next(t)(i) = r11 or
next(t)(i) = r12 or next(t)(i) = r13 or
next(t)(i) = r14 or next(t)(i) = r15 or
next(t)(i) = r16 or next(t)(i) = r17 or
next(t)(i) = r18 or next(t)(i) = r19 or
next(t)(i) = r20)
C. The Readers/Writers Model in SPIN, SAL, and PVS

\[ V_9(t) : \text{bool} = \exists (i: \text{index}) \cdot (i = pD(t) \land (\text{next}(t)(i) = r3 \lor \text{next}(t)(i) = \text{rlseAtPwr} \lor \text{next}(t)(i) = r4 \lor \text{next}(t)(i) = r6 \lor \text{next}(t)(i) = r7) \lor r6 \lor \text{rdcnt}(t) > 0) \]

\[ V_{10}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land (\text{next}(t)(i) = \text{rlseAtPwr}) \implies (\text{rd}(t) = 0 \land \text{cnt}(v(t)) < 1) \]

\[ V_{11}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land (\text{next}(t)(i) = \text{rlseAtPvr} \lor \text{next}(t)(i) = r2 \lor \text{next}(t)(i) = r3 \lor \text{next}(t)(i) = \text{rlseAtPvr} \lor \text{next}(t)(i) = r4 \lor \text{next}(t)(i) = \text{rlseAtPvr} \lor \text{next}(t)(i) = r7 \lor \text{next}(t)(i) = r9)) \implies \text{rd}(t) = \text{rdcnt}(t) \]

\[ V_{12}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land (\text{next}(t)(i) = r1 \lor \text{next}(t)(i) = \text{rlseAtPvr} \lor \text{next}(t)(i) = r2 \lor \text{next}(t)(i) = r4 \lor \text{next}(t)(i) = \text{rlseAtPvr} \lor \text{next}(t)(i) = r7 \lor \text{next}(t)(i) = r9)) \implies \text{rd}(t) = \text{rdcnt}(t) \]

\[ V_{13}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land (\text{next}(t)(i) = r3 \lor \text{next}(t)(i) = \text{rlseAtPvr} \lor \text{rd}(t) = \text{rdcnt}(t) - 1) \]

\[ V_{14}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land \text{next}(t)(i) = r6) \implies \text{rd}(t) = \text{rdcnt}(t) + 1 \]

\[ V_{15}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land \text{next}(t)(i) = \text{rlseAtPvr}) \implies (\text{wt}(t) = 0 \land \text{cnt}(v(t)) < 1) \]

\[ V_{16}(t) : \text{bool} = \exists (i: \text{index}) : (i = pD(t) \land (\text{next}(t)(i) = v3 \lor \text{next}(t)(i) = v2)) \implies (\text{wt}(t) = 1 \land \text{cnt}(v(t)) < 1) \]