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

Vera Pantelic, Xiao-Hui Jin[1], Mark Lawford, and David Parnas[2]
Department of Computing and Software, McMaster University
1280 Main St W., Hamilton, ON, Canada L8S 4K1
Telephone: +1 (905) 525–9140 ext.27406, Fax: +1 (905) 524–0340
E-mail: {pantelv} *at* mcmaster *dot* ca, {Xiaohui *dot* Jin} *at* Inf *dot* ethz *dot* ch, {lawford, parnas} *at* mcmaster *dot* ca

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 [1]. The process involved describing the program by a table and then performing a manual "column-by-column" inspection for safety and clean completion properties. The key step in the process is obtaining an invariant strong enough to prove the properties of interest. This paper presents partial automation of this approach. 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 automated theorem proving together with model checking on the earlier model to 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 [1], discovering several random and one systematic mistake of the proof. Subsequently, a new, significantly automated proof was performed.

Keywords—Concurrency, SAL, PVS, invariant, tables.

1. Introduction

A reliable and effective inspection approach for the inspection of concurrent programs is proposed in [1]. Inspection is made easier and reliable by inspecting each of the components separately. Further, each component's behavior is described using program function tables [2]. However, as will be shown in this paper, the manual proof of the correctness criterion given in [1] did not consider the whole transition relation described by the program function table. Automated tool support helped to discover the flaws of the manual proof easily.

There are many approaches to mechanized formal analysis of concurrent systems represented with transition relations. Those include theorem proving, model checking, abstraction and model checking, automated abstraction, bounded model checking [3], [4], and equivalence verification [5], [6], [7].

Model checking is a technique for verifying finite state concurrent systems [8]. 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 arbitrary number of processes.

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, currently it requires substantial human guidance.

This paper represents an extension of the approach of [1], providing partial automation of the proposed inspection process. The original program can be analyzed in SPIN for a small number of processes. SPIN is a model checking tool specialized for handling concurrent systems. Its specification language provides the primitives for interprocess communication [9]. Model checking in SPIN can be particularly useful for the 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 [1]. The transition relation of the concurrent system as given by the table is rewritten into the SAL model checker [10] and model checked for safety and liveness properties. However, at this point, SAL does not support tables, nor does it offer a full typechecker. The table is then rewritten into the PVS specification language [10]

[1]Present affiliation: Software Engineering, ETH Zentrum, RZ F9, Clausiusstrasse 59, 8092 Zurich, Telephone: +41 44 632 8296, Fax: +41 44 632 14 35

[2]McMaster Emeritus, present affiliation: Department of Computer Science and Information Systems, Faculty of Informatics and Electronics, University of Limerick, Limerick, Ireland, Telephone: (353) 61 202731, Fax: (353) 61 202734

table construct and checked for consistency and completeness. Safety properties are proved in PVS using the inductive invariant approach [3]. 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. However, few properties are inductive. Failed proof goals suggest the auxiliary invariants that we then use to strengthen the initial property. Then, we try to prove that the strengthened invariant (the 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 the liveness property then requires additional strengthening of the inductive invariant found during the proof of the safety property.

2. Preliminaries

2.1 Introduction to the Approach

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:

a) Auxiliary variables are introduced to capture all the information needed to analyze the program, specify the control flow.
b) The requirements of the program are formulated as a mathematical specification.
c) The primitive operators are specified (e.g., synchronization primitives) — this should have been done before the program was written.
d) 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.
e) The program is described in a tabular representation.
f) 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.

2.2 Example Application: Readers/Writers Problem

A classic concurrency problem is the Readers/Writers problem [11]. Two different kinds of processes, readers and writers, access the common resource. An unlimited number of readers can concurrently access the resource, but a writer must have exclusive access to the resource. Among two variants of this problem presented in [11], the one that gives readers priority over the writers is chosen. In this case, the readers' preference is weak.

2.2.1 The Original Program: The program used to solve the chosen variant from [11] is reproduced below with shared variable definitions followed by the reader process on the left, and the writer process on the right:

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

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

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

$$(rd = 0 \vee wt = 0) \wedge wt < 2 \tag{1}$$

2.2.2 Applying the proposed approach to the example application: Applying the steps of the proposed approach (as described in the Section 2.1), the original Readers/Writers program can be partially rewritten as in Figure 1. The complete code for a reader is given in [1].

The *stop* symbol tells us when an executing process can be interrupted, allowing other processes to resume their execution, i.e., each line of Figure 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 $waitAtPm1$, $rlseAtPm1$, $waitAtPm2$, $rlseAtPm2$,

```
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
```

Fig. 1. Readers/Writers program rewritten

$waitAtPwr$, $rlseAtPwr$, $waitAtPww$, $rlseAtPww$ 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 [1].

The program is then rewritten into a main table and subtables, parts of which are reproduced in Figure 2. The complete original main table containing 41 columns and all associated subtables can be found in [1]. In the tables M is the total number of processes and n is the number of reader processes with $0 \leq n \leq M$. A parameter k ($0 < k \leq M$) is introduced to identify a representative process. The variable pID is used to identify the currently executing process. Two additional boolean expressions are introduced: $IsReader(k)$ and $IsWriter(k)$, that stand for $0 < k \leq n$ and $n < k \leq M$, respectively. The interested reader is referred to [1] for the details on rewriting the program as in Figure 1 to the table as in Figure 2. The program state can be described as a 7-tuple $(rdcnt, rd, wt, mutex, w, next, pID)$.

2.2.3 Showing Clean Completion: We say that a program has a clean completion when all of its constituent processes can finish their 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), a vector of decreasing quantity DQ is defined in [1]:

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

where M is the total number of processes, $Pros$ is the number of the processes that have not reached the ***EOP*** (End of Program) label yet, and $IntRW$ is the function mapping all the values of $next$ to integers, as indicated in Table I, as taken from [1].

Let $l = 1, 2$ be program states. Then $next_l$ and $Pros_l$ are the values of $next$ and $Pros$ in state l. As before, *n* is the number of the reader processes ($0 \leq n \leq M$). Let

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

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

$$DQ_l = (Pros_l, IntRW(next_l[i]), \ldots, IntRW(next_l[i]))$$

TABLE I

THE *IntRW* FUNCTION DEFINITION

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

Then, the order property of DQ is given by the Table II (taken from [1]) where ***DQorder*** stands for $DQ_1 > DQ_2$.

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.

3. Specification and Analysis in SPIN

SPIN is a model checker specialized for modeling and analysis of concurrent systems [9]. SPIN supports rendezvous and buffered message passing, and communication through shared memory. We use it to formalize and model-check the original version (before it is rewritten into a table) of the Readers/Writers concurrent program with a fixed number of readers and writers.

The semaphores used for synchronization in the Reader/Writer problem are easily modeled with the help of SPIN's rendezvous port feature.

	$`pID = k \wedge IsReader$					
	$`next[k] = r1$			$`next[k] = waitAtPm1$	$`next[k] = rlseAtPm1$	...
	$`m.cnt > 1$	$`m.cnt = 1$	$`m.cnt < 1$			
$rdcnt' =$		$`rdcnt$	$`rdcnt$	$`rdcnt$	$`rdcnt$	...
$rd' =$		$`rd$	$`rd$	$`rd$	$`rd$	
$wt' =$		$`wt$	$`wt$	$`wt$	$`wt$	...
$m.cnt' =$		$`m.cnt - 1$	$`m.cnt - 1$	$`m.cnt$	$`m.cnt$	...
$m.set'\| =$		$m.set' = `m.set$	$m.set' = `m.set \cup \{k\}$	$m.set' = `m.set$	$m.set' = `m.set$	...
$w.cnt' =$		$`w.cnt$	$`w.cnt$	$`w.cnt$	$`w.cnt$	...
$w.set'\|$		$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$	$w.set' = `w.set$	...
$pID'\|$		$next[pID']' \neq EOP$	$next[pID']' \neq EOP$	$next[pID']' \neq EOP$	$next[pID']' \neq EOP$	...
$next'\|$		Tab2	Tab3	$\forall j : next[j]' = `next[j]$	Tab5	...
	1	2	3	4	5	...

Tab2: $\forall j$,

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

$\vdots$

Fig. 2. Part of the tabular specification of the Readers/Writers program

The safety property defined as

$$(rd = 0 \vee wt = 0) \wedge wt < 2 \wedge rd \geq 0 \wedge wt \geq 0 \qquad (2)$$

is to be checked in SPIN. We note that the safety property as given here is a modified version of the property defined in Equation 1 (originally taken from [1]). Since the *rd* and *wt* variables are integers, adding the last two conjuncts as in Equation 2 requires that the number of readers/writers cannot be negative). The safety property is easily formalized using LTL logic or `never` claim [9].

The check of clean completion can be automatically done in SPIN by checking for the absence of invalid end states.

However, checking the properties even for the system of 10 readers and 10 writers is very slow (longer than 20 hours). We can use the SPIN's approximation techniques described in [9] (collapse compression, bitstate hashing, hash-compact) to make a quick check, but these techniques do not guarantee 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.

In summary, we found it easy to formalize and verify limited instances of the original Readers/Writers problem in SPIN. This check could be useful for refutation purposes: some potential bugs of a program can be discovered in this early stage of the verification. However, only instances of the system with a relatively small number of processes can be verified.

4. Formalization in PVS and SAL

After analysis in SPIN, 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 any errors introduced when the specification was rewritten. We will use the SAL model as a prelude to theorem proving in PVS of the general model with an arbitrary number of readers and writers (as will be shown in the next section). During the theorem proving process, every potential auxiliary invariant found by PVS is model checked in SAL.

The formalization in PVS is given first, as the analysis in PVS has the central spot in our approach. The formalization in SAL is then given by comparing it with PVS formalization. The details can be found in [13].

4.1 Formalization in PVS

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 [10].

The `decl` theory with the definitions of types, functions, *etc.* for the readers/writers problem with an arbitrarily large number of processes is given in Figure 3.

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

TABLE II

THE ORDER PROPERTY OF DQ

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

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

Fig. 3. Theory `decl`

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 [1]). A global variable of the type `state` contains the resources shared by all the processes: semaphores `m` and `w`, then counters `rd`, `wt`, `rdcnt`, the array of processes' labels `next`, and `pID`, the identifier of the currently executing process. Indices of the array are the process identifiers. The predicate `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` from Table I, used for proving the clean completion of the program.

The tabular representation of the Readers/Writers rewritten program is represented as a theory in PVS. The table is modeled with a transition relation `trans` (the interested reader is referred to [13] for details).

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

4.2 Formalization in SAL

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 [12]. The current SAL toolset provides explicit state, symbolic, bounded, infinite bounded and witness model checkers for SAL [10]. We will use the symbolic model checker called *sal-smc*, which uses linear temporal logic (LTL) as its assertion language.

As the specification language of SAL has input syntax similar to PVS, we found rewriting the tabular specification into the SAL specification language to be straightforward. However, 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. Moreover, 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. Furthermore, the `pID` is not modeled as the part of the global state, as it is implicitly modeled by model checker.

5. Verifying the Manual Proof

The safety requirement as stated in Equation 1 is defined in PVS. Strictly following the manual proof of [1], we first try to prove the first conjunct of Equation 1, by proving that it is true after initialization and, row by row (or, column by column, for the original table), that it is preserved after every statement in the program. Before being checked in the theorem prover, every new, auxiliary invariant, as found in the manual proof, is model checked in the SAL model-checker for a specific instance of the problem. This check is automatic and fast.

The proposed combination of theorem proving and model checking discovered several random and one systematic mistake in the manual proof. More precisely, model checking itself indicated that some of the invariants found in the manual proof were not invariants of the program. Theorem proving offered further insight into the depth of the systematic mistake made: it pointed to the sources of the errors. By investigating the manual proof, we came to the conclusion that the random errors were made because some branches of the proof were not explored at all (the interested reader is referred to [13] for details). Furthermore, proving the auxiliary invariants of the form $(\exists i : (i = pID(t) \ \wedge \ next(t)(i) = l)) \Rightarrow P(t)$, where P is a predicate on the global state of the system t, and l is some label, discovered a more serious flaw of the proof: only part of the transition relation was explored. The manual proof actually considered the relation from the table with an additional assumption: the `pID` of the currently executing process does not change after the transition of the program to the next state.

6. Verification in PVS Revisited

In this section we describe a significantly automated proof of the safety property. 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.

We formalize a lemma per invariant. Using the knowledge gained from the analysis in the previous section, we designed a strategy to prove these lemmas, or, rather, gain new invariants. Branches of the proof corresponding to the invariants that are universally quantified on `i` (the identification number of a process) are split into two cases. When the process of interest is the executing one (`i!1 = pID`, where `i!1` is a term used to instantiate an invariant), we apply `GRIND`, and contemplate the invariants from the unprovable sequents. However, we choose to skip the case when the process of interest is not the one executing (`i!1 /= pID`), since the vast majority of the failed goals corresponding to this branch can be subsumed into two invariants saying that there cannot be more than one process in the critical section of a semaphore (`m` or `w`) at a time — we will refer to those invariants as "semaphore" invariants. We continue on with strengthening the property using the same tactic without using "semaphore" invariants, until we prove that the conjunction of the global property and the newly found invariants is inductive for the branches corresponding to `i!1 = pID`. 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 becomes the conjunction of the old and newly found ones. However, we choose to prove only the properties that were not proved (for `i!1=pID`) in the previous iteration and the newly found ones.

As indicated in step 3, all the auxiliary invariants are first model checked. 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 written use a "divide and conquer" policy: every proof is split into 31 branches (where 31 is the number of non-blank table columns) are then tackled with the same tactic. This tactic is chosen so that the degree of the automation 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).

7. Proving Clean Completion

The clean completion property says that all the processes will eventually complete, i.e., reach the label `EOP`. However, model checking this property (under the assumption of weak fairness of the scheduler) in SAL runs out of memory. Therefore, we prove the liveness property for a system with fixed number of readers/writers as suggested in Section 2.2.3. We prove the theorems `dqa`, `dqb`, and `dqc` corresponding to the first, second, and third part, respectively, of the theorem of decreasing quantity. However, we found no need to define a DQ vector as suggested in Section 2.2.3 (originating from [1]), because of the assumption that no new readers/writers arrive after the initialization of the system. Moreover, if `Pos` is defined as the number of the reader/writer processes with a label other than `EOP`, then the case of a process reaching the label `EOP` ($\texttt{Pos}_1 > \texttt{Pos}_2$) can be considered as the case of decreasing one of the components of the vector IntRW defined as:

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

Therefore, the vector $IntRW$ can be used as the decreasing quantity. We say that $IntRW$ has decreased if there is at least one element of the $IntRw$ that has decreased, while all the others have decreased or remained the same. Note, however, that the ordering defined by `DQdecrease` is not total. We later prove that this ordering implies the order originally formulated in [1].

The theorem `dqb` is not expressible in LTL logic, 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 system. If we bring the insight of our problem into it, we are able to formalize it in LTL. Moreover, a module

is introduced to cope with the size of automata generated in model checking the theorems `dqa` and `dqb` [13].

In PVS, proving the `dqb` theorem required additional strengthening of the invariant that was found sufficient for proving the safety property. This reduction of the state space 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 [1].

8. Model Checking Results

Tables III and IV show the model checking results for SPIN and SAL, respectively. SAL performs worse than SPIN,

TABLE III

SPIN MODEL CHECKING RESULTS

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

TABLE IV

SAL MODEL CHECKING RESULTS

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

due in part to the relatively higher complexity of SAL model and the greater size of state variable vector. Model checking, however, was not sufficient to prove the correctness of the systems with arbitrary number of processes. In both cases, only specific instances of the system with a small number of processes could be checked.

Complete SPIN, PVS, and SAL files are available from the first author on request.

9. Conclusions

We provided partial automation of the inspection process of [1]. We believe that many of the issues dealt with in the analysis of the Readers/Writers example in this paper 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) used, or the invariants of the form $\forall i : (next(t)(i) = l \Rightarrow P(t))$, where P is a predicate on the global state of the system t, and l is some label, for which the written strategies can be reused to a certain extent).

In our verification of the Readers/Writers program we used a specific implementation of semaphore. 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 published in [14], against a particular implementation.

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 existential abstraction as suggested in [15]. Even with the aid of automation, proving such programs remains a lot of work that would only be justified for critical programs. We believe that further research is needed to introduce more automation and make the procedure applicable to a broader class of problems.

Acknowledgment

The authors would like to thank Ryszard Janicki and Sanzheng Qiao for reviewing Vera Pantelic's thesis.

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] V. Pantelic, "Inspection of concurrent systems: Combining tables, theorem proving and model checking," Master's thesis, McMaster University, December 2005.

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

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