Assertion-based proof checking of Chang-Roberts leader election in PVS

Assertion-Based Proof Checking of Chang-Roberts Leader Election in PVS*

Judi Romijn[1], Wieger Wesselink[1], and Arjan Mooij[2]

[1] Dept. of Mathematics and Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
`{j.m.t.romijn,j.w.wesselink}@tue.nl`
[2] School of Computer Science, The University of Nottingham
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, United Kingdom
`arjan.mooij@cs.nott.ac.uk`

Abstract. We report a case study in automated incremental assertion-based proof checking with PVS. Given an annotated distributed algorithm, our tool ProPar generates the proof obligations for partial correctness, plus a proof script per obligation. ProPar then lets PVS attempt to discharge all obligations by running the proof scripts.

The Chang-Roberts algorithm elects a leader on a unidirectional ring with unique identities. With ProPar, we check its correctness with a very high degree of automation: over 90% of the proof obligations is discharged automatically. This case study underlines the feasibility of the approach and is, to the best of our knowledge, the first verification of the Chang-Roberts algorithm for arbitrary ring size in a proof checker.

1 Introduction

Checking proofs with proof assistant tools is a recognized and valuable activity. Manual proofs tend to have small mistakes, and sometimes more serious ones. Tools such as PVS provide powerful generic strategies, but the interaction with these is usually rather involved. For correctness of distributed systems, the tedious and cumbersome task of proof checking may benefit greatly from automation if there are general proof structures that apply and if a formalization of the specification language is available. E.g., the TAME strategies [1] for PVS relieve the book keeping in interactive proof checking for I/O automata.

In [15], we introduced a front-end tool supporting assertion-based verification in the style of Owicki and Gries [18]. Given an annotated program, our tool ProPar (Proof checking of Parallel programs) generates the proof obligations for partial correctness, plus a proof script per obligation. It feeds the resulting specification to the proof checker PVS [19], which attempts to discharge each proof obligation by running the supplied proof script.

There is a growing interest to use general purpose provers like PVS as a back-end for dedicated proof checking tasks. For example, [10] discusses

* This research was supported by the NWO under project 016.023.015: "Improving the Quality of Protocol Standards".

customizations of and extensions to PVS that would support this, while [16] describes a tool that supports batch proving using PVS.

Closer to our work is [17,20], in which the Owicki-Gries theory is formalized in the Isabelle prover. Their work is mainly theoretical, while we aim at supporting the proof checking of concrete examples in an effective way, and at dealing with the incremental nature of constructing a correct annotation.

This paper reports a case study: we apply our tool ProPar to the ring leader election algorithm by Chang and Roberts [3]. Our interest in the correctness of this algorithm is twofold. For this simple algorithm, some of the correctness proofs require rather involved reasoning on the ring structure, making it a non-trivial case study. In addition, to the best of our knowledge, no existing proof for this protocol for arbitrary ring size has ever been checked mechanically.

We construct a correct annotation of the algorithm in several steps, while illustrating the use of ProPar and PVS. Of the final annotation, over 90% of the proof obligations has been discharged automatically. In four cases we have to supply a proof in PVS ourselves, only two of these proofs are non-trivial.

Compared to [15,13], the effectiveness and user-friendliness of ProPar has been improved greatly, according to the findings of this case study.

Overview. This paper is organised as follows. In Section 2, we recall the Owicki-Gries theory and explain ProPar. In Section 3, the Chang-Roberts algorithm is introduced, and related work on correctness of this algorithm is discussed. Sections 4, 5 and 6 present the actual annotation of Chang-Roberts and our ProPar/PVS efforts. Section 7 has the conclusions and future work.

2 The Owicki-Gries Theory and the Tool ProPar

We check the correctness of annotated programs with the tool ProPar[1] (Proof checking of Parallel programs), which we introduced in [15]. ProPar takes an annotated program as input, and generates proof obligations for the PVS proof checker[2] [19] for local and global correctness of the annotation. Per proof obligation, ProPar generates a proof script to enable running PVS in batch mode.

2.1 Annotated Programs

ProPar accepts the following language constructs in annotated programs:

– empty statement (**skip**)
– sequential composition of two or more statements (. . . **;**. . .)
– alternative selection of one or more guarded statements (**if** . . . **fi**)
– repetition of one or more guarded statements (**do** . . . **od**)
– parallel composition over the elements of a type (**par** x: . . . **rap**)

[1] ProPar is available from the authors upon request.
[2] ProPar can also generate Isabelle output, but that feature was not exploited here.

– parallel composition of two or more statements[3] (**co** ... **oc**)
– atomic statements

Multiple guarded statements are separated by []; a guard is separated from the corresponding statement by $\rightarrow$. Atomic statements are statements whose execution cannot be interfered by executions of statements in other processes. An example is the multiple assignment $x, y := a, b$. It is up to the user to determine the atomic statements, and to model them in the language of the prover.

Programs can be annotated using assertions, that may be placed at the control points of a program, i.e. at locations right before or after a statement. Assertions are predicates on the state of the program. For repetitions and parallel compositions there is also a notion of invariants. An invariant must be placed at the control point i before keyword **do** (resp **par**), and is equivalent to an assertion placed at i and at all control points within the repetition (resp. parallel composition). This gives the user a convenient shorthand and allows ProPar to combine many duplicate proof obligations.

2.2 Proof Obligations

An annotated program is to be considered correct if all assertions are correct. An assertion at a control point is correct if the state of the program satisfies the assertion, whenever execution is at the control point. Note that termination of programs is not considered. The tool ProPar automatically generates proof obligations from which the correctness of a program can be derived.

All proof obligations are expressed in terms of Hoare triples. A Hoare triple $\{P\}\ S\ \{Q\}$ is a boolean that is *true* if and only if each terminating execution of statement S that starts from a state satisfying predicate P is guaranteed to end up in a state satisfying predicate Q. The weakest liberal precondition $wlp.S.Q$, is the weakest precondition P such that $\{P\}\ S\ \{Q\}$ is a correct Hoare triple. More formally $\{P\}\ S\ \{Q\} \equiv [P \Rightarrow wlp.S.Q]$, where $[\ldots]$ is a shorthand for "for all states", i.e. a universal quantifier binding all free variables.

According to Owicki-Gries [18] an assertion Q in a process is correct iff:

– *local correctness*: If Q is an initial assertion, Q is implied by the precondition of the program. If Q is preceded by atomic statement $\{P\}\ S$ (with P an assertion preceding statement S), then $\{P\}\ S\ \{Q\}$ is a correct Hoare triple.
– *global correctness*: For each atomic statement $\{P\}\ S$ in a different process, $\{P \wedge Q\}\ S\ \{Q\}$ is a correct Hoare triple.

ProPar obtains the proof obligations for local correctness by rewriting (parts of) the annotated program according to Table 1, while applying each line as rewrite rule from left to right. The rewriting ends when no statements remain. Nested parallel compositions are treated seamlessly in this manner. The Hoare

[3] A **co** statement can be easily expressed in terms of **par**. However, in many cases the number of proof obligations is smaller for **co**.

Table 1. The local correctness proof obligations per statement type

$$\begin{array}{ll}
\{P\}\text{skip}\{R\} & \Leftarrow [P \Rightarrow R] \\
\{P\}\text{atomic-statement-S}\{R\} & \Leftarrow [P \Rightarrow \text{wlp.atomic-statement-S}.R]
\end{array}$$

$$\{P\}S_0;\{Q\}S_1\{R\} \quad \Leftarrow \begin{cases} \{P\}S_0\{Q\} \\ \{Q\}S_1\{R\} \end{cases}$$

$$\begin{array}{l}
\{P\} \\
\textbf{do } B_0 \rightarrow \{P_0\}S_0 \\
[]\ B_1 \rightarrow \{P_1\}S_1 \\
\textbf{od} \\
\{R\}
\end{array} \quad \Leftarrow \begin{cases} [P \wedge B_0 \Rightarrow P_0] \\ [P \wedge B_1 \Rightarrow P_1] \\ [P \wedge \neg(B_0 \vee B_1) \Rightarrow R] \\ \{P_0\}S_0\{P\} \\ \{P_1\}S_1\{P\} \end{cases}$$

$$\begin{array}{l}
\{P\} \\
\textbf{if } B_0 \rightarrow \{Q_0\}S_0 \\
[]\ B_1 \rightarrow \{Q_1\}S_1 \\
\textbf{fi} \\
\{R\}
\end{array} \quad \Leftarrow \begin{cases} [P \wedge B_0 \Rightarrow Q_0] \\ \{Q_0\}S_0\{R\} \\ [P \wedge B_1 \Rightarrow Q_1] \\ \{Q_1\}S_1\{R\} \end{cases}$$

$$\begin{array}{l}
\{P\} \\
\textbf{par } x : \\
\quad \{Q_0.x\}S.x\{Q_1.x\} \\
\textbf{rap} \\
\{R\}
\end{array} \quad \Leftarrow \begin{cases} [\forall x : P \Rightarrow Q_0.x] \\ \forall x : \{Q_0.x\}S.x\{Q_1.x\} \\ [(\forall x : Q_1.x) \Rightarrow R] \end{cases}$$

$$\begin{array}{l}
\{P\} \\
\textbf{co} \\
\quad \textbf{proc } \{P_0\}S_0\{Q_0\} \textbf{ corp} \\
\quad \textbf{proc } \{P_1\}S_1\{Q_1\} \textbf{ corp} \\
\textbf{oc} \\
\{R\}
\end{array} \quad \Leftarrow \begin{cases} [P \Rightarrow P_0] \\ [P \Rightarrow P_1] \\ \{P_0\}S_0\{Q_0\} \\ \{P_1\}S_1\{Q_1\} \\ [(Q_0 \wedge Q_1) \Rightarrow R] \end{cases}$$

triples corresponding to global correctness are related to atomic statements, and can therefore be expressed directly in wlps of atomic statements.

To illustrate the proof obligations generated by ProPar, we consider the program fragment of the parallel composition in Table 1, where we assume that assertions P, Q_0, Q_1 and R are placed at labels 0, 1, 2 and 3. The three local correctness proof obligations on the right hand side are encoded in PVS as follows, where we assume that S is an atomic statement:

loc_Q0_stat_0: **lemma**
  forall $(s : \textit{state})$: **forall** $(x : X)$: $\textit{lab\_0}(s) \Rightarrow Q0(x)(s)$
loc_Q1_stat_1: **lemma**
  forall $(s : \textit{state})$: **forall** $(x : X)$: $\textit{lab\_1}(x)(s) \Rightarrow \text{wlp\_}S(x)(Q1(x))(s)$
loc_R_stat_2: **lemma**
  forall $(s : \textit{state})$: (**forall** $(x : X)$: $\textit{lab\_2}(x)(s)) \Rightarrow R(s)$

Here, the variable s ranges over all possible program states, and is introduced to model the brackets [. . .] in Table 1. The logical variable lab_i represents the conjunction of the assertions located at the control point with label i.

2.3 Proof Scripts

ProPar generates PVS proof scripts for each of the generated proof obligations, and writes them to a .prf file in the PVS proof format. PVS is then run in batch execution mode to check whether each proof obligation is discharged.

As the first step in a proof script, ProPar automatically selects assertions and invariants from relevant control points and inserts them with the `lemma` command. Here, ProPar also instantiates quantifier variables for the state with appropriate skolem variables. Contrary to the previous version, ProPar now always explicitly chooses the skolem variables itself. Experience has taught us that the automatic choices of PVS are not always suitable.

The second step is a simplification step, in which commands like `replace`, `assert` and `simplify` are used. We now discuss an example of the difficulties encountered when trying to automate this step. Consider the proof state

```
{-1}  FORALL (c: component): lab_6(c)(s!1) => inv_0a(s!1)
[-2]  FORALL (c: component): lab_6(c)(s!1)
  |-------
[1]   ass_7a(s!1)
```

Given the assumption that type *component* is non-empty, one would expect there are high-level commands available that simplify this into

```
{-1} inv_0a(s!1)
[-2] FORALL (c: component): lab_6(c)(s!1)
  |-------
[1]  ass_7a(s!1)
```

However, we did not succeed in generating proof scripts that achieve this simplification in all the different contexts that may occur. We solved this problem by introducing two custom lemmas as follows, with t a generic type:

quantifier_lemma_1: **lemma**
$\textbf{forall}\ (P : \mathit{bool}) : (\textbf{forall}\ (x : t) : P) = ((\exists (x : t) : \mathit{true}) \Rightarrow P)$
quantifier_lemma_2: **lemma**
$\textbf{forall}\ (P : [t \to \mathit{bool}], Q : [t \to \mathit{bool}]) : (\textbf{forall}\ (x : t) : P(x)) \Rightarrow$
$(\textbf{forall}\ (x : t) : Q(x)) = \textbf{forall}\ (x : t) : P(x) \Rightarrow Q(x)))$

This is a generalization of the approach we used in previous versions [15,13].

The third and last part of a proof script consists of a generalization of the `grind` command to perform the hard work. This is the same as in [15].

2.4 User Input

The input of ProPar consists of

– An annotated program, written in a prover independent language.
– An import file containing prover-dependent definitions.
– A file containing proof hints (optional, see below).
– A file containing manual proofs (optional, see below).

The annotated program contains only references to assertions, guards and wlp's of atomic statements. In the import file, the user has to express these in the language of the prover. Usually this is a straightforward task.

Proof Hints. When the automatic proof of a lemma fails, the user may provide proof hints, i.e. high-level directions to help the prover for a certain lemma. Compared to [15], the use of proof hints is now much more generic. Proof hints apply to the sequence of lemmas introduced in the beginning of a proof script. If ProPar introduces too many lemmas, the prover becomes very inefficient. The order of the lemmas can also influence the performance of the prover. Moreover, additional lemmas from other theories may be needed. Through proof hints the prover can focus on the appropriate lemmas, in the optimal order.

Finally, if all else fails, the user can supply a manual proof.

3 The Chang-Roberts Leader Election Algorithm

The leader election algorithm introduced by Chang and Roberts in [3] is designed for a uni-directional ring consisting of components with unique identities. A strict total order $>$ on the identities is assumed. The algorithm elects the greatest identity present according to $>$.

Each component may send its own unique identity to its neighbour in the ring. Components only forward messages with identities which are greater than any identity received thus far. The component that receives its own identity concludes that its identity is the greatest in the ring and wins the election.

The algorithm was intended as an improvement on the leader election part of Le Lann's token passing algorithm [8]. Here, due to faulty connections, multiple elections can overlap and correctness is not guaranteed. Like [3], we restrict ourselves to reliable connections and one election round only.

3.1 The Algorithm in Assertion-Based Style

In Figure 1, the Chang-Roberts leader election algorithm is shown, including assertions for the correctness specification. The program is a parallel composition of the repetition to be executed by each component. The program terminates only when all components have finished the repetition. The labels 0a, 0b, etc. that precede assertions are generated by ProPar, we stick to that labelling in the remainder of this paper. Likewise, we refer to the guards in the selection statement at location 2 as guard 2a, 2b and 2c.

```
   var $leader : [comp \to nat]$,   $ready : [comp \to bool]$
0: {ass 0a: $(\forall_{c:comp} : leader_c = id(c))$}
   {ass 0b: $(\forall_{c:comp} : \neg ready_c)$}
   par $(c : comp)$:
1:     do $\neg ready_c \ \to$
2:         if $ready_{prev(c)} \ \to$
3:             $ready_c := true$
           [] $leader_{prev(c)} = id(c) \ \to$
4:             $ready_c := true$
           [] $leader_{prev(c)} > leader_c \ \to$
5:             $leader_c := leader_{prev(c)}$
           fi
       od
6: rap
7: {ass 7a: $(\forall_{c_1,c_2:comp} : leader_{c_1} = leader_{c_2}$}
   {ass 7b: $(\forall_{c_1,c_2:comp} : leader_{c_1} \geq id(c_2))$}
   {ass 7c: $(\exists_{c:comp} : leader_c = id(c)$}
```

Fig. 1. The Chang-Roberts leader election algorithm

Assertions 0a, 0b, 7a, 7b and 7c express the correctness of the algorithm. These enforce that upon termination of the parallel statement, all components have elected the same leader, i.e. the greatest identity present in the ring.

To start, each component has its own identity for leader. Inside the repetition at control point 1, the election takes place. Each component c monitors the leader identity of its immediate neighbour $prev(c)$ in the unidirectional ring. When the neighbour's leader identity is greater than the component's own leader identity, guard 2c evaluates to true, and the component may copy it in statement 5. In this manner, candidate leader identities spread over the ring, until they are overtaken by a better identity. The greatest identity spreads over the ring until it reaches the originating component. Then this component signals it has won because guard 2b evaluates to true. At this point, the election is finished.

A component can only terminate the repetition after its *ready* flag has been set to true. In this way the algorithm ensures that all components can find out that the election has ended, an aspect that is often ignored in the literature.

We model communication by having the receiver poll the sender's current leader identity. This is clearly equivalent to synchronous communication with explicit messages. Moreover, for this particular algorithm, a version with asynchronous communication simulates our polling version: the sending of a message in the asynchronous case can be related to the polling moment in our version.

Ring structure. We assume the type *comp* for the components on the ring, and a constant function $id : [comp \to nat]$ mapping each component to its unique identity which is a natural number. In this way we immediately have the total order $>$. For the ring structure, we assume the constant function $prev : [comp \to comp]$ which points at the predecessor of a component in the

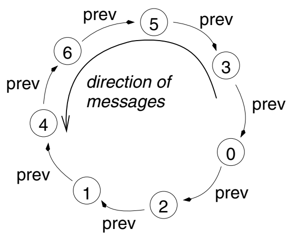

Fig. 2. A unidirectional ring

ring. An example ring is shown in Figure 2. Here, the unique identity of a component is in the node, and the arrows between components indicate the predecessor relation.

3.2 Related Work

Lynch et al. have studied Chang-Roberts in the I/O automata language. [12] has a correctness proof, but proofs are only sketched. In [11,9] a performance analysis of a timed version is given but the proofs are not checked (in contrast to other results in both publications). An IOA model is online at the IOA homepage (`http://theory.csail.mit.edu/tds/ioa/`). In [7], a finite instance of this IOA model is checked in Isabelle/IOA.

Garavel et al. [6] have model checked Le Lann's full token passing algorithm for concrete ring sizes in the formal language LOTOS and proposed an improvement. They find mistakes in the presence of faulty connections that lead to overlapping election rounds, which is outside the scope of our case study. However, the mistakes found and the improvement proposed suggest that having a proper notification of the election termination should be part of the protocol. We have this in our version of the leader election, as explained above.

Sen [21] reports on another model checking experiment in his thesis, comparing the tool POTA to SPIN.

A correctness proof for arbitrary ring size is given by Chen et al in [4] in the formal language μCRL, but it has not been proof checked in any tool. In fact, this paper turns out to contain some essential mistakes in both specification and proof, thus underlining the need for machine-checked verifications.

Many papers study the performance of Chang-Roberts and related algorithms (like [2]). That is outside the scope of this paper.

Apparently, despite the attention in the literature, the Chang-Roberts algorithm has not been verified mechanically for arbitrary ring size. It seems that this algorithm, while small and simple, has the appropriate degree of complexity to make it a nice proof checking challenge for our tool ProPar.

4 A Correct Chang-Roberts Annotation (Phase 1)

We must extend the annotation given in Figure 1 with assertions and invariants until it is provably correct, i.e. each proof obligation generated by ProPar is discharged either automatically by running PVS on the generated proof script, or manually by proving it in PVS ourselves. In the coming sections, we extend the annotation and then run ProPar to see how far PVS gets in batch mode.

In the first annotation step, we work from the annotation in Figure 1 towards the one in Figure 3 (new parts marked $*$).

We start by adding invariants. We weaken assertion 7b to a local version in invariant 0a. The two are equivalent when assertion 7a holds. Assertion 7a becomes true when at least one component signals that the election is finished. We state this in invariant 0b with no more than two quantified variables, using transitivity of $=$. When exiting the parallel statement, by successful termination of each repetition, the premise in invariant 0b holds for all components hence establishing assertion 7a. For assertion 7c, we observe that each leader identity stored by a component is the identity of some component in the ring. We state this as invariant 0c. Combining assertion 7a and invariant 0c yields assertion 7c.

To establish the correctness of these invariants, we add assertions to the control points inside the parallel statement. We start by stating each selection guard as an assertion following the guard's execution. We can do so if the guard continues to hold regardless of what the other components do. For each of the guards 2a, 2b and 2c this is indeed the case. Similarly, we state the negation of the repetition guard at location 6.

In addition, we state in assertion 4b that the current component's leader identity is equal to its neighbour's leader identity. We also state in assertion 5b that none of the components have finished the repetition.

4.1 ProPar Results

We run the ProPar tool on the files containing the annotated algorithm from Figure 3. ProPar generates 36 proof obligations and proof scripts. Of these, 29 are discharged automatically by PVS. The seven remaining obligations are:

– local correctness of assertion 4b (when executing guard 2b),
– local correctness of assertion 7c (when exiting the parallel statement), and
– global correctness of assertions 4a and 4b when executing assignment 5,
– global correctness of assertion 5b when executing assignment 4,
– global correctness of invariant 0b when executing assignment 3 or 4.

For assertion 4b, when guard 2b holds, clearly c still has its own identity for leader. If this was not the case, then it copied a better identity from its predecessor, and then guard 2b cannot hold. Reasoning about this requires information on the leader identities that a component has had between the first (its own) and the last (the winner). We can do so by adding a history variable, which is discussed in the next section. We choose this as our next move, in the hope that it will help in discharging the other proof obligations.

```
var $leader : [comp \to nat]$,   $ready : [comp \to bool]$
0: {ass 0a: $(\forall_{c:comp} : leader_c = id(c))$}
   {ass 0b: $(\forall_{c:comp} : \neg ready_c)$}
   {inv 0a: $(\forall_{c:comp} : leader_c \geq id(c))$}                                   *
   {inv 0b: $(\forall_{c_1,c_2:comp} : ready_{c_1} \Rightarrow leader_{c_1} = leader_{c_2})$}   *
   {inv 0c: $(\forall_{c_1:comp} : (\exists_{c_2:comp} : leader_{c_1} = id(c_2)))$}              *
   par $(c : comp)$:
1:     do $\neg ready_c \to$
2:         if $ready_{prev(c)} \to$
3:             {ass 3a: $ready_{prev(c)}$}                                              *
               $ready_c := true$
           [] $leader_{prev(c)} = id(c) \to$
4:             {ass 4a: $leader_{prev(c)} = id(c)$}                                     *
               {ass 4b: $leader_c = leader_{prev(c)}$}                                  *
               $ready_c := true$
           [] $leader_{prev(c)} > leader_c \to$
5:             {ass 5a: $leader_{prev(c)} > leader_c$}                                  *
               {ass 5b: $(\forall_{c_1:comp} : \neg ready_{c_1})$}                           *
               $leader_c := leader_{prev(c)}$
           fi
       od
6:     {ass 6a: $ready_c$}                                                              *
   rap
7: {ass 7a: $(\forall_{c_1,c_2:comp} : leader_{c_1} = leader_{c_2})$}
   {ass 7b: $(\forall_{c_1,c_2:comp} : leader_{c_1} \geq id(c_2))$}
   {ass 7c: $(\exists_{c:comp} : leader_c = id(c))$}
```

Fig. 3. Chang-Roberts algorithm (phase 1, marked $*$)

In order to maintain global correctness of invariant 0b under assignment 4, we calculate the wlp. We find a stronger version of assertion 4b: when guard 2b holds, it is in fact the case that the winning identity has traversed the entire ring is the leader for each component. This is added as assertion 4c in the following section. This assertion will help in discharging the remaining global correctness obligations for assertions 4a, 4b and 5b. Note that reasoning to show local correctness of the new assertion 4c requires induction on the ring structure. Here, the history variable can be useful too.

5 A Correct Chang-Roberts Annotation (Phase 2)

The first step for dealing with the remaining proof obligations from Section 4.1, for which PVS cannot successfully execute the ProPar proof script, is to introduce a history variable $leaders_c$ for each component c in the ring. The history variable records all the values that the program variable $leader_c$ takes on during execution. This enables us to compare current and past leader identities of a component and its neighbour. We can add any history variable if it does not change the program's behaviour.

We now state invariants 0d to 0g, expressing that each new leader identity accepted by component c is better than c's own identity, has been copied from c's predecessor, and hence must be in the $leaders_{prev(c)}$ collection.

The new annotation is Figure 4 with everything from the previous annotation (unmarked) and the new parts added in this phase (marked $*$). Note that the part marked $**$ belongs with the final annotation (see Section 6). The two annotations are merged in Figure 4 to save space.

As announced, we also add assertion 4c which helps to maintain invariant 0b under assignment 4, and which implies assertion 4b. Assertion 4a is weakened to make use of variable *leaders*, by combining guard 2b and invariant 0d.

For history variable *leaders*, we ensure the proper initial value with the new assertion 0c. Its value is updated in statement 5: whenever a new leader identity is copied it is also added to the *leaders* collection. Invariants 0d to 0g express the additional information that we require for proving correctness.

5.1 ProPar Results

We run ProPar on the incremented annotated algorithm from Figure 4 (except the part marked $**$). Of the 52 proof obligations generated, PVS discharges 46 automatically through the ProPar proof scripts.

We notice that PVS fails for local correctness of assertions 5b and 7a whereas it ran successfully for the previous annotation. PVS gets confused by the growing number of applicable lemmas: this annotation has seven invariants instead of three. If we give ProPar proof hints for these failing proof, we can easily generate the successful script again. We supply only the invariants from the previous annotation plus the assertions of the location prior to the current statement:

```
loc_ass_5b_stat_2: lab_2_inv_0a lab_2_inv_0b lab_2_inv_0c
loc_ass_7a_stat_6: lab_6_ass_6a lab_6_inv_0a lab_6_inv_0b lab_6_inv_0c
```

With these proof hints, the proofs generated by ProPar are accepted by PVS.

Of the 52 proof obligations, PVS has now 48 discharged automatically. The four obligations for which PVS fails are ($*$ marks the new obligation):

– local correctness of assertion 4b (when executing guard 2b),
$*$ local correctness of assertion 4c (when executing guard 2b),
– local correctness of assertion 7c (when exiting the parallel statement), and
– global correctness of invariant 0b when executing assignment 3.

Apparently, all previous global correctness obligations are now discharged except the one for invariant 0b under assignment 4, assertions, and only one of the new proof obligations remains unproved.

We establish local correctness of assertion 7c with local correctness of assertion 7a and invariant 0c as proof hints. Since 7a and 7c are at the same control point, with 7a preceding 7c, we can use local correctness of 7a as a lemma here. We adjusted ProPar to allow such proof hints.

For local correctness of assertion 4c, we need one final invariant. This is described in the following section.

```
    var $leader : [comp \rightarrow nat]$,   $leaders : [comp \rightarrow setof(nat)]$,   $ready : [comp \rightarrow bool]$
0:  {ass 0a: $(\forall_{c:comp} : leader_c = id(c))$}
    {ass 0b: $(\forall_{c:comp} : \neg ready_c)$}
    {ass 0c: $(\forall_{c:comp} : leaders_c = \{leader_c\})$}
    {inv 0a: $(\forall_{c:comp} : leader_c \geq id(c))$}
    {inv 0b: $(\forall_{c_1,c_2:comp} : ready_{c_1} \Rightarrow leader_{c_1} = leader_{c_2})$}
    {inv 0c: $(\forall_{c_1:comp} : (\exists_{c_2:comp} : leader_{c_1} = id(c_2)))$}
    {inv 0d: $(\forall_{c:comp} : leader_c \in leaders_c)$}                                        $*$
    {inv 0e: $(\forall_{c:comp,n:nat} : n \in leaders_c \Rightarrow leader_c \geq n)$}               $*$
    {inv 0f: $(\forall_{c:comp} : (leaders_c - \{id(c)\}) \subseteq leaders_{prev(c)})$}              $*$
    {inv 0g: $(\forall_{c:comp,n:nat} : n \in leaders_c \Rightarrow n \geq id(c))$}                  $*$
    {inv 0h: $(\forall_{c_1,c_2:comp} : leader_{c_2} = id(c_1)$
                 $\Rightarrow (\forall_{c_3:comp} : mp(c_1,c_3) \leq mp(c_1,c_2) \Rightarrow c_1 \in leaders_{c_3}))$}   $**$
    par $(c : comp)$:
1:      do $\neg ready_c$ $\rightarrow$
2:          if $ready_{prev(c)}$ $\rightarrow$
3:              {ass 3a: $ready_{prev(c)}$}
                $ready_c := true$
            [] $leader_{prev(c)} = id(c)$ $\rightarrow$
4:              {ass 4a: $id(c) \in leaders_{prev(c)}$}                                            $*$
                {ass 4b: $leader_c = leader_{prev(c)}$}
                {ass 4c: $(\forall_{c_1:comp} : leader_{c_1} = leader_c)$}                         $*$
                $ready_c := true$
            [] $leader_{prev(c)} > leader_c$ $\rightarrow$
5:              {ass 5a: $leader_{prev(c)} > leader_c$}
                {ass 5b: $(\forall_{c_1:comp} : \neg ready_{c_1})$}
                $leader_c, leaders_c := leader_{prev(c)}, leaders_c \cup \{leader_{prev(c)}\}$      $*$
            fi
        od
6:      {ass 6a: $ready_c$}
    rap
7:  {ass 7a: $(\forall_{c_1,c_2:comp} : leader_{c_1} = leader_{c_2})$}
    {ass 7b: $(\forall_{c_1,c_2:comp} : leader_{c_1} \geq id(c_2))$}
    {ass 7c: $(\exists_{c:comp} : leader_c = id(c))$}
```

Fig. 4. Chang-Roberts algorithm (phase 2 marked $*$, and phase 3 marked $**$)

6 A Correct Chang-Roberts Annotation (Phase 3)

We add a final invariant to enable the manual proof for local correctness of assertion 4c. The annotation obtained in this way is Figure 4, now including the line marked $**$. Invariant 0h expresses for component c_2 that has the identity of c_1 for leader, that all components on the predecessor path from c_2 to c_1 have also seen the identity of c_1. Note the use of function $mp(c_1, c_2)$ which computes the distance in the ring between c_1 and c_2 by counting the number of *prev* steps back from c_2 to c_1.

6.1 ProPar Results

Local correctness of assertion 7b is lost, and mended with proof hints, as in Section 5.1. Of the 55 current proof obligations, PVS discharges 51 automatically, 4 of these through our use of proof hints. The obligations for which PVS fails are ($*$ marks the new obligation):

– local correctness of assertion 4b (when executing guard 2b),
– local correctness of assertion 4c (when executing guard 2b),
– global correctness of invariant 0b when executing assignment 3.
$*$ global correctness of invariant 0h when executing assignment 5.

Only the second obligation seems to require an involved proof, but the others turn out to be too tricky for PVS with ProPar proof scripts, even with our proof hints. All of these require a manual proof.

We run ProPar on the annotation of Figure 4 with the proof hints and manual proofs (discussed in the remainder of this section), to find that all proof obligations are discharged, hence correctness of Chang-Roberts is now established.

6.2 Manual Proofs

When starting a manual proof, the proof script generated by ProPar is very helpful. We use the PVS option to step through the generated proof, until we reach the point where ProPar calls on the semidecision procedures (like `grind`). Here we take over and manually instruct PVS step by step until we have `Q.E.D.`

We have created a theory for the ring structure with domain-specific knowledge. In this theory, we have proved some useful lemmas which are used in the manual proofs for invariant 0b and assertion 4c.

The proofs for local correctness of assertion 4b and for global correctness of invariant 0b under statement 3 are easy. Global correctness of invariant 0h under statement 5 requires a complicated case distinction on the three quantified variables but this is very manageable.

As announced, for local correctness of assertion 4c we must use some kind of induction on the ring structure. We apply `measure-induct+` and use as measure function the distance $mp(c_1, c_2)$. The base case is easy. For the induction step, based on the assumption that we have the same leader for c_1 and $prev(c_2)$ we can then prove the induction step using all invariants except 0b. This requires a complicated proof of about 90 PVS proof steps.

7 Conclusions and Future Work

We successfully applied our method to check correctness of the Chang-Roberts algorithm for arbitrary ring size, by creating a proof in an incremental and automated fashion. In the end, 51 out of 55 proofs were handled automatically. Five of the 51 proofs required straightforward proof hints from the user. In addition, four manual proofs were needed, two of which were rather involved.

The most significant parts of the proof effort were developing a correct annotation, and manually proving the remaining proof obligations. Other activities (creating an import file, creating proof hints and interacting with the tool) were negligible compared to these.

During the case study we made pragmatic improvements to ProPar, that resulted in a higher rate of automatically handled proofs, without changing anything in the theory. E.g., the proof hints mechanism now allows for user-defined lemmas to be introduced, and for local correctness results of pre-assertions at the current control point to be exploited. Custom lemmas are applied to smoothen the quantifications encountered which differ for control points in the body versus the control point at the end of a parallel composition. In addition, the proof scripts have been changed to make them behave in a more predictable way, by explicitly instantiating with skolem variables.

7.1 PVS Discussion

In contrast to manual proofs, when dealing with large numbers of generated proofs, it is essential that proof strategies are robust and behave in a predictable way. In our previous case study [15] we discovered a bug in PVS that prevented certain proofs to be completed automatically. This has been repaired since version 4.0 (PVS bug 920). In this case study another bug has been encountered in PVS 4.0 and submitted (PVS bug 979, reportedly fixed but yet not released).

In some cases unpredictable behavior of PVS caused problems. For example, certain proofs suddenly failed after hiding an unnecessary antecedent. Still many seemingly simple proof obligations exist that PVS cannot handle automatically. To deal with this, we introduced new proof hints, and applied custom lemmas to support the prover.

A more powerful proof script language for PVS is desirable. For example, it would be useful to be able to enumerate possible instantiations of quantifier variables, and to specify in which order PVS should try to use them. PVS commands like `use` and the higher level `grind` often choose the right instantiations, but for proofs in batch mode "often" is not good enough.

7.2 Future Work

There are several directions for future work. Termination of programs (or more generally progress conditions [5]) could be supported by generating proof obligations for the decrease of a user supplied norm function. Another topic is to study to what extent inductive proofs (e.g., for security protocols like in [14]) can be supported, if the user supplies a measure function. Soundness could be warranted by automatically verifying that the generated proof obligations are sufficient, similar to the approach in [17].

Finally, PVS has weak support for controlling instantiations of quantifier variables, which sometimes causes the tail part of the generated proofs to fail. Until PVS is improved in this respect, we can possibly circumvent this by arranging for the user to supply suitable instantiation hints.

References

1. Archer, M., Heitmeyer, C., Riccobene, E.: Proving invariants of I/O automata with TAME. Automated Software Engineering 9(3), 201–232 (2002)
2. Chan, M.Y., Chin, F.Y.L.: Optimal resilient distributed algorithms for ring election. IEEE Trans. on Parallel and Distributed Systems 4(4), 475–480 (1993)
3. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema finding in circular configurations of processes. Comm. of the ACM 22(5), 281–283 (1979)
4. Chen, T., Han, T., Lu, J.: Analysis of a leader election algorithm in μCRL. In: Proceedings of CIT 2005, pp. 841–847. IEEE Computer Society, Los Alamitos (2005)
5. Dongol, B., Mooij, A.J.: Progress in deriving concurrent programs: emphasizing the role of stable guards. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 140–161. Springer, Heidelberg (2006)
6. Garavel, H., Mounier, L.: Specification and verification of various distributed leader election algorithms for unidirectional ring networks. Science of Computer Programming 29(1–2), 171–197 (1997)
7. Hamberger, T.: Integrating theorem proving and model checking in Isabelle/IOA. Technical Report TUM-I, T.U. Munich (1999)
8. Le Lann, G.: Distributed systems - towards a formal approach. In: 1977 IFIP Congress Proceedings, Information Processing, vol. 77, pp. 155–160. North-Holland, Amsterdam (1977)
9. Luchangco, V.: Using Simulation to Prove Timing Properties. PhD thesis, Massachusetts Institute of Technology (1995)
10. Lüttgen, G., Muñoz, C., Butler, R., Vito, B.D., Miner, P.: Towards a customizable PVS. Technical Report ICASE 2000-4, CR-2000-209851. NASA Langley (2000)
11. Lynch, N.A.: Proving performance properties (even probabilistic ones). In: Proceedings of FORTE 1994, pp. 3–20. Chapman and Hall (1995)
12. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
13. Mooij, A.J.: Constructive formal methods and protocol standardization. PhD thesis, Technische Universiteit Eindhoven (2006)
14. Mooij, A.J.: Constructing and reasoning about security protocols using invariants. In: Proceedings of REFINE 2007, ENTCS. Elsevier, Amsterdam (to appear, 2007)
15. Mooij, A.J., Wesselink, J.W.: Incremental verification of Owicki/Gries proof outlines using PVS. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 390–404. Springer, Heidelberg (2005)
16. C. Muñoz. Batch proving and proof scripting in PVS. Technical Report NIA 2007-03, CR-2007-214546, NASA Langley (2007)
17. Nipkow, T., Prensa Nieto, L.: Owicki/Gries in Isabelle/HOL. In: Finance, J.-P. (ed.) FASE 1999. LNCS, vol. 1577, pp. 188–203. Springer, Heidelberg (1999)
18. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6, 319–340 (1976)
19. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
20. Prensa Nieto, L.: Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL. PhD thesis, T.U. Munich (2002)
21. Sen, A.: Techniques for Formal Verification of Concurrent and Distributed Program Traces. PhD thesis, The University of Texas at Austin (2004)