Automated verification of Owicki/Gries proof outlines comparing PVS and Isabelle

Koudijs, Johan C.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Automated verification of Owicki/Gries proof outlines: comparing PVS and Isabelle

by

J.C. Koudijs

Supervisors: dr. J.M.T. Romijn
ir. A.J. Mooij
dr.ir. J.W. Wesselink

Eindhoven, January 2006

Preface

This report results from my graduation project(s) at the Department of Mathematics and Computer Science of Eindhoven University of Technology.

As introduction, I have worked on a case study of a UPnP Power Management protocol, under supervision of Judi Romijn. I have made various models of UPnP environments, and checked certain properties using model checking techniques. Being a bit disappointed by the limits of model checking, and being up for a new challenge, I decided to head a different direction after this introduction.

The remaining nine months I have explored the wonderful world of theorem provers — which was new to me — and worked on a project which was eventually titled "Automated verification of Owicki/Gries proof outlines: comparing PVS and Isabelle". This second project was under supervision of Arjan Mooij and Wieger Wesselink.

This second (main) project is the subject of this report. The model checking case study has hardly any relation to this subject, and its results are described in Appendix A.

I would like to thank a few people for their involvement and support. First, I would like to express my gratitude to my father, Johan Koudijs Sr., and my mother, Cilia Koudijs-de Kok. I also thank Arjan, Judi, and Wieger for their time, support, and (personal) interest.

Johan C. Koudijs
December, 2005

Contents

Chapter 1

Introduction

Parallel programs have become increasingly popular and they have found their way into a large number of systems and areas. Due to the complexity of parallel programs, they are difficult to design and it is difficult to verify their correctness formally. The risk of failing to meet the requirements is however often not an acceptable one.

State-space exploration methods like model checking are often used for the verification of parallel programs. One of the advantages of model checking is that it can be automated. But, model checking suffers from the so called state-space explosion problem and is therefore limited to parallel programs that have a finite state-space that is not larger than resources allow. In practice, state-spaces are often too large or infinite.

Another method for the verification of parallel programs is annotating the program with assertions and prove the correctness of these assertions. The theory of Owicki/Gries [OG76] can then be used to reduce the verification problem of the annotated program (i.e. proof outline) to a limited number of proof obligations. An advantage of this method is that it enables proving more general properties, and it does not suffer from the state-space explosion problem. The major disadvantages are that inventing the appropriate annotation is a challenging task, and that in general human effort is required to prove the theorems.

Typically even small parallel programs have a large number of proof obligations. It requires a considerable amount of effort to prove the proof obligations manually. Many of them are trivial, and their proof is often omitted in manual proofs. Humans tend to be sloppy, stubborn and overconfident; this is often the cause of mistakes, errors, and ultimately, disasters.

Theorem provers are available to assist the user in proving the proof obligations. They can check the proof of the user, and depending on the theorem prover used, they can automate (part of) the proof of proof obligations. Theorem provers can reduce the amount of human effort required for the proof of theorems considerably, and they increase the confidence in the results of the proofs.

Mooij and Wesselink have developed a tool [MW05] that, given an annotated parallel program, generates the proof obligations in the specification language used by the PVS theorem prover [ORS92]. Also, the tool generates default proof scripts for the proof obligations. Using these proof scripts, large numbers of proof obligations can be proved completely automatically with PVS. Applied to their real-life case study of a distributed spanning tree algorithm [MW03], which is a complex algorithm, they were able to prove almost 90% of the proof obligations fully-automatically. The remaining 10% of the proof obligation are proved interactively with PVS.

The PVS theorem prover was chosen as back-end of the tool because it is one of the most popular theorem provers available. [GH98] suggests that the level of automation offered by the interactive theorem provers PVS and Isabelle [Pau94] is comparable. The goals of the research discussed in this report are to investigate how the Isabelle theorem prover performs on this kind of automated verification and to determine how effective PVS and Isabelle are with respect to automated verification. Thereto support for Isabelle as back-end of the tool has been added, automatic proof strategies for Isabelle have been developed, and we have experimented with the verification of several parallel programs.

1.1 Owicki/Gries proof outlines

A parallel program consists of a number of sequential programs to be executed concurrently. Execution of a sequential program (or component) results in a process that consists of a sequence of atomic actions. Execution of the parallel program results in an interleaving of the processes.

The state of a parallel program consists of the values of the variables of the program. An assertion is a predicate over this state, and is placed between two consecutive atomic actions in the components. We denote assertion P as $\{P\}$. An assertion is correct if the state of the parallel program satisfies the predicate whenever the component is at the point in the sequential program where the assertion is located. An annotated program (or proof outline) is a program annotated with assertions. The annotation of such a program is correct if all assertions are correct.

A Hoare-Triple [Hoa69] $\{P\}S\{Q\}$ is a boolean that has the value *true* if each terminating execution of statement S that starts from an initial state satisfying P is guaranteed to end up in a final state satisfying Q. This only expresses partial correctness, because termination of statements is not considered.

The theory of Owicki/Gries [OG76] is used to determine the correctness of assertions. This theory states that an assertion is correct if it is both *locally* correct and *globally* correct. An assertion P is *locally* correct if it is either

- an initial assertion of a component and is implied by the pre-condition of the program, or
- if P is established by the preceding atomic action S with pre-assertion Q (denoted as $\{Q\}S$), i.e. $\{Q\}S\{P\}$ is *true*.

An assertion P is *globally* correct (or interference free) if for each atomic action $\{Q\}S$ taken from another component, $\{P \wedge Q\}S\{P\}$ is *true*.

1.2 Theorem proving

Theorem provers are computer programs that assist the user in proving theorems. They provide a specification language to be used to specify proof obligations, and they define which inference rules can be used to prove the proof obligations. Theorem provers are often categorised according to their level of automation.

Proof checkers are theorem provers that check detailed proofs supplied by the user and offer no automation. Automatic theorem provers prove theorems fully-automatically. The user specifies the proof obligation, initiates the proving process and waits until the theorem prover concludes with the answer "yes, this is correct", "no, this is not correct", or "I cannot decide whether this is correct". Interactive theorem provers combine automated reasoning with user interaction. The user gives hints to the system to instruct which direction the proof attempt should head. Depending on the theorem prover and the proof obligation, proofs can still be completely or partly automated.

PVS and Isabelle are interactive theorem provers, and their proof process is comparable. The user supplies a proof obligation (lemma), and the theorem prover maintains a proof state, which consists of subgoals that are still to be proved. Initially this proof state consists of exactly one subgoal: the original proof obligation. The theorem prover provides tactics that allow the user to apply valid inference rules to the proof state. Tactics range from the application of basic inference steps, to application of fully-automated proof strategies that attempt to finish the proof. Tactics may split goals into (smaller) subgoals, subgoals that are trivially valid are removed from the proof state, and the original proof obligation is proved to be valid once we arrive at a proof state without any subgoals.

Prototype Verification System (PVS)

The interactive theorem prover PVS (Prototype Verification System) is being developed at SRI Computer Science Laboratory. The specification language of PVS is highly expressive and is based

on typed higher-order logic. PVS has been applied to many nontrivial problems. NASA Langley is one of the main sponsors and users of PVS.

Goals in PVS are represented as sequents, which are of the form:

$$\begin{array}{l} A_1 \\ \vdots \\ A_m \\ \hline \vdash \\ C_1 \\ \vdots \\ C_n \end{array}$$

where $0 \leq m$ and $0 \leq n$. We often use the linear representation:

$$A_1 , \ldots , A_m \quad \vdash \quad C_1 , \ldots , C_n$$

This goal states that we have to prove that the conjunction of the antecedent formulas $A_1 \ldots A_m$ implies the disjunction of the consequent formulas $C_1 \ldots C_n$. Thus the logical interpretation is:

$$A_1 \wedge \ldots \wedge A_m \quad \Rightarrow \quad C_1 \vee \ldots \vee C_n$$

PVS uses a sequent calculus as underlying logic. In addition to many basic inference steps available to the user, there are tactics available that facilitate automated reasoning. This includes a term rewriter, a BDD simplifier and powerful (arithmetic) decision procedures. We consider PVS version 3.2 in this report.

Isabelle

The interactive theorem prover Isabelle is being developed at the university of Cambridge and the technical university of Munich. Isabelle is generic and flexible, different logics can be used and users can add new logics and tactics. In this document we consider Isabelle 2004 in combination with the logic HOL, the Higher Order Logic. When referring to Isabelle, from now on, we actually refer to the combination Isabelle/HOL. The Archive of Formal Proofs [AFP] is a collection of formalisations and verifications using Isabelle.

The specification language of Isabelle is based on functional programming languages. Isabelle uses a meta language to represent goals. Goals are of the form:

$$[\![A_1 , \ldots , A_n]\!] \quad \Longrightarrow \quad C$$

which states that C is to be proved using the assumptions $A_1 \ldots A_n$. The logical interpretation of this goal is:

$$A_1 \wedge \ldots \wedge A_n \quad \Rightarrow \quad C$$

Tactics in Isabelle are mostly based on term rewriting, unification and application of inference rules like natural deduction. The automated tactics of Isabelle mostly combine these three and add backtracking and tableaux methods to find a proof for subgoals. The user can add rules to the set of default rules used by the automated tactics.

In Chapter 4 we give a much more extensive introduction to Isabelle.

1.3 Related work

It is typically a challenging and time consuming task to invent the appropriate annotation, and prove the annotation afterwards. [FvG99] shows that the theory of Owicki/Gries can be used to *design* parallel programs (and their annotation) from their formal specification. This way the resulting annotation is correct by *construction*.

We are not the first to use a theorem prover to verify Owicki/Gries proof outlines. The most-related work is [PN02], in which a formalisation of Owicki/Gries in the theorem prover Isabelle is introduced and discussed. The most significant differences between this formalisation and [MW05] are that [MW05] focuses on the incremental verification of the annotation, and that [MW05] puts more emphasis on fully automated verification.

We are also not the first to compare PVS and Isabelle/HOL. In [GH98], PVS and Isabelle/HOL are compared. They give a consumers' report, based on their experiences with the prover. One of their conclusions is that the level of automation offered by PVS and Isabelle is comparable.

1.4 Overview

The remainder of this report is structured as follows. We use the tool described in [MW05] to generate the proof obligations and their proof scripts. In Chapter 2 this tool is discussed.

In Chapter 3 we discuss how programs and their annotation can be modelled and verified with the tool.
The tool is extended with support for the Isabelle theorem prover as back-end. In Chapter 4 we give a short introduction to Isabelle and we discuss automated verification with Isabelle in detail. We discuss the changes to the tool, the modelling of the programs and proof obligations in Isabelle, the automated Isabelle proof strategies, and the results of experiments with Isabelle.
We have experimented with automated PVS and Isabelle verification of several relatively small algorithms. The results of these experiments are discussed in Chapter 5.
Many proof obligations contain universal and existential quantifiers; Chapter 6 gives an overview of the options we have in PVS and Isabelle proofs dealing with quantifications.
In Chapter 7 the verification of a larger algorithm is discussed: a Garbage Collection Algorithm from [PN02].

Finally, in Chapter 8 we conclude this report.

Chapter 2

Proof generator

2.1 Introduction

In [MW05] a tool is described which, given an annotated parallel program (proof outline), generates input files for the PVS theorem prover. The output of the tool consists of the proof obligations for the annotation. The theorem prover PVS can then be used to prove the proof obligations. Many of them can be proved fully-automatically using default proof scripts generated by the tool, and those that cannot be proved automatically can be proved interactively (manually) with the theorem prover.

We use this tool to generate the proof obligations and proof scripts for our verifications. In this chapter we give an overview of the relevant aspects and features of the tool. This overview is strongly based on [MW05].

The tool is a Python script that reads an annotated parallel program, and generates the input files for the theorem prover PVS. The environment of the tool is depicted in Figure 2.1.

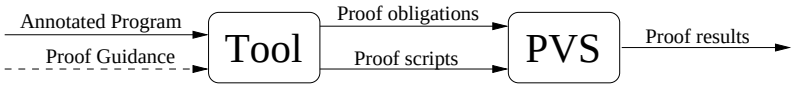

Figure 2.1: Environment of the tool

In Section 2.2 we discuss how annotated parallel programs are represented in the tool and which language constructs are available. The generated proof obligations are discussed in Section 2.3 and the default PVS proof script is discussed in Section 2.4. The proofs are executed in batch-mode. Proof obligations that cannot be proved automatically, or that are aborted because they cannot be proved within an acceptable period of time, require proof guidance from the user. In Section 2.5 we discuss how the user can influence the proof scripts generated by the tool.

2.2 Annotated program

As example program we use a parallel linear search algorithm from [MW05]. Given a number of boolean functions on naturals, this algorithm searches for a value that is mapped to the value *true* by one of these functions. The program is depicted in Figure 2.2. The upper part of the figure consists of the declaration of the variables of the parallel program, and their types. The values of the variables represent the state of the parallel program.

A parallel program consists of a (possibly dynamic) number of components. The type *component* represents the set of component identifiers of the example program. For each of the

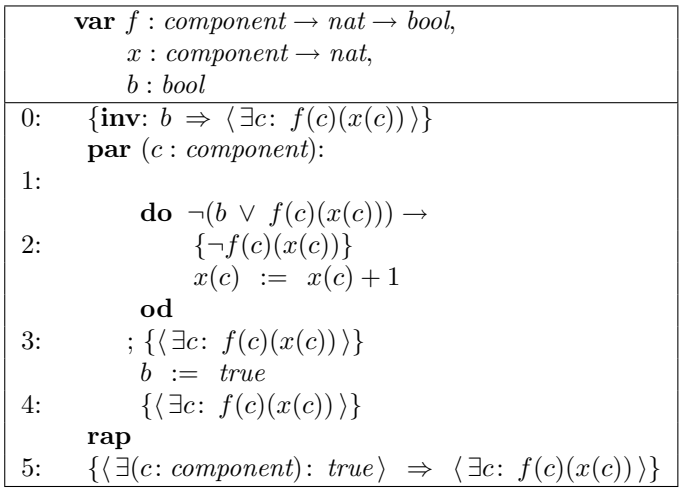

Figure 2.2: Parallel linear search

functions on naturals, a component which is dedicated to this function is created. For each component with identifier c, $f(c)$ represents the corresponding function and $x(c)$ is a private variable of the component. Boolean variable b is shared by the components.

The lower part of the figure contains the annotated parallel program. Each assignment (**:=**) is an atomic statement; other atomic statements that we will frequently encounter are:

- Multiple assignment: $x, y := X, Y$. X and Y are evaluated, and then their values are assigned to variables x and y, respectively.
- **skip**. An empty statement, it does nothing.

Composite statements combine atomic actions into larger statements. The program consists of a number of composite statements:

- Parallel composition over the elements of a type: **par**. In the example, executing the **par** statement creates zero or more processes: for each element c of *component* a process is created for the component with identifier c (the body of the **par** statement).
- Repetition of a statement: **do**
- Sequential composition of two statements: **;**

One type of composite statement that is not used in this example is the *Alternative construct* or **if**-statement. This statement introduces non-deterministic selection. It consists of one or more guards (predicates over the state), and each guard is associated with a statement. The guards are evaluated repeatedly by the component executing this statement, until one of the guards evaluates to *true*. Once one of the guards evaluates to *true*, the corresponding statement is executed.

A control point is a location between two consecutive atomic actions of a component. The numbers that are followed by a colon are labels that identify the control points. An assertion is a predicate over the state of the system and is located at a control point. The assertion is correct if, whenever a component is at this control point, the state of the system satisfies the predicate. Multiple assertions can be placed at a control point; such a sequence of assertions denotes their conjunction.

Invariant I, denoted as $\{\textbf{inv}: I\}$, is a predicate over the state of the system which can be placed at a control point just before a parallel composition, and is an abbreviation of an assertion that is also placed at every control point within the parallel composition. The control point with label 0 is a special control point: the assertions and invariants at this first control point also represent the pre-condition of the program.

Annotated programs are modelled using two files. One file defines the structure of the program. This structure is completely independent of the theorem prover and is used to generate the proof obligations in the language of the prover. Another file models the types, the annotation, the guards and the atomic statements in the specification language of the theorem prover. The generated proof obligations depend on this file.

Hoare-triples for atomic statements are defined using their weakest liberal pre-condition (*wlp*). The weakest liberal pre-condition of an atomic statement S is a predicate transformer $wlp.S$. The $wlp.S$ of a predicate Q, denoted as $wlp.S.Q$, is the weakest condition P such that $\{P\}S\{Q\}$ is *true*, i.e.

$$\{P\}S\{Q\} \quad \equiv \quad [\, P \;\Rightarrow\; wlp.S.Q\,]$$

where the square brackets are a shorthand notation for "for all states". Atomic statements are modelled as their *wlp*, and the (atomic) guards are modelled as predicates over the state of the system.

2.3 Proof obligations

The theory of Owicki/Gries is used for the verification of the annotated parallel programs. Using this theory, the problem of verifying the correctness of the annotation of the program is reduced to verifying a number of proof obligations in the form of Hoare-triples. The proof obligations are only concerned with the atomic actions; composite statements are decomposed into atomic actions.

[MW05] puts emphasis on *incremental* verification and construction of the annotation. We do not focus on incremental verification, but the proof obligations are modelled in a special way to facilitate this incremental verification. Before discussing an example of a proof obligation, we first give some motivation behind the way the proof obligations are modelled. We restrict this to those aspects that are relevant for this report, and we refer to [MW05] for additional information.

To facilitate incremental verification it is important that old successful proof scripts remain successful after adding new assertions. This is achieved by ensuring that there are no textual changes in the old proof obligations. To that end, the assertions are fully decoupled from each other. Consider a proof obligation for local correctness, which is of the form:

$$[\, P \wedge Q \;\Rightarrow\; Z\,]$$

with predicates P, Q and Z. Using the principle of indirect inequality we can decompose this, and obtain:

$$\langle\, \forall X\colon\; [\, X \Rightarrow P \wedge Q\,] \quad \Rightarrow \quad [\, X \Rightarrow Z\,]\,\rangle$$

and because implication is conjunctive in its consequent we obtain:

$$\langle\, \forall X\colon\; [\, X \Rightarrow P\,] \wedge [\, X \Rightarrow Q\,] \quad \Rightarrow \quad [\, X \Rightarrow Z\,]\,\rangle$$

This can be modelled as proof obligation $[\, X \Rightarrow Z\,]$, after declaring a fresh dummy variable X as a logical variable and introducing the two axioms $[\, X \Rightarrow P\,]$ and $[\, X \Rightarrow Q\,]$. If, later on, assertion R is added and the proof obligation is weakened into $[P \wedge Q \wedge R \;\Rightarrow\; Z\,]$, then only an additional axiom stating $[\, X \Rightarrow R\,]$ needs to be added, and the correctness of the old proof script for $[\, X \Rightarrow Z\,]$ cannot be endangered as long as all used axioms are employed explicitly.

Instead of introducing new variables for each proof obligation, the structure of the theory of Owicki/Gries is exploited. Namely, the antecedent of each proof obligation is the conjunction of all assertions at one or two control points. For each control point with label i, a logical variable *lab_i* is introduced, which is related to the assertions and invariants that hold on control point i, using axioms. Similarly, for each control point with label i where a parallel composition starts, a variable *scp_i* is introduced, and an axiom relates the *scp* variable to the control points within this parallel composition.

Scope lemmas are defined to prove the relation between the *scp* variables and invariants, much like the relation between *lab* variables and assertions. They are used for the proof of the invariance

of invariants. The proof of scope lemmas is trivial, and they can be proved automatically with PVS using the introduced axioms.

The following is the proof obligation for local correctness of the assertion at control point 4 of the example program, modelled in PVS:

```
loc_ass_4a_stat_3: lemma
    forall (s : state):
        forall (c : component): lab_3(c)(s) ⇒
            wlp_stat_3(c)(ass_4a(c))(s)
```

This proof obligation should be interpreted as:

- For each state s,
- and for each component c that is at control point 3,
- executing the statement at control point 3 establishes ass_4a (first post-assertion).

The axiom that defines the relation between *lab_3* and the assertions at control point 3 are available (and are required) for the proof of this proof obligation. *wlp_stat_3* models the weakest liberal pre-condition of the statement at control point 3, and *ass_4a* models the assertion at control point 4. Both are defined in PVS by the user, and all other definitions for this proof obligation are generated by the tool.

2.4 Proof scripts

```
(skosimp* :preds? t)

(lemma "lab_2_ass_2a")
(inst -1 "s!1" "c!1")
... ...

(branch (grind :if-match nil)
    ((then (try (reduce) (fail) (skip))
        (then (inst? :if-match all) (then (reduce :if-match all) (fail) ) ) )) )
```

Figure 2.3: Default PVS proof script

It is important to have a generic proof script that can prove as many proof obligations as possible, with a minimum amount of manual interaction. Figure 2.3 shows the default proof script for the proof obligations in PVS. This proof script is applied to each type of proof obligation by default and relies heavily on the automation offered by the theorem prover. It consists of three parts:

1. The first strategy decomposes the top-level structure of the proof obligation and introduces skolem constants and type constraints for the bound variables, i.e. the state and the component identifiers.
2. The axioms that define the relation between the *lab* variables and the assertions are employed, and known constants are substituted.
3. Automated strategies are applied in an attempt to finish the proof. First "grind" without quantifier instantiation is applied. "grind" is a catch-all strategy, it repeats strategies such as simplification using decision procedures, term rewriting and BDD simplification. They are repeated until all subgoals are proved or none of the strategies have any effect. "reduce"

is the main workhorse of "grind", and is applied to the subgoals that remain, using heuristic quantifier instantiation. If this does not complete the proof, then repeatedly all instantiations of a bound variable are substituted, and "reduce" is tried again.

2.5 Proof guidance

The proofs are executed in batch-mode with PVS. Proof obligations that cannot be proved automatically using the default proof script, or cannot be proved in a reasonable time period, need guidance from the user. Obviously, we would like to limit this to a minimum.

There are two ways to influence the proof scripts of the proof obligations. The first is by using proof hints. Proof hints are used by the user to specify which of the available assertions are relevant for a proof obligation. This can reduce the search space of the tactics considerably. Often, the developer of the algorithm can easily indicate which assertions are relevant for a proof, while this can be very difficult to determine by a theorem prover.

If a proof obligation cannot be proved using the default proof script in combination with proof hints from the user, then the user has no other choice but to prove it manually (interactively) using the theorem prover. If the user succeeds at proving the proof obligation with the theorem prover, the proof script used for this proof can be supplied to the tool, and the tool will use this proof script for this proof obligation, from that point on.

Chapter 3

Modelling parallel programs

3.1 Introduction

The goals of this chapter are to show how parallel programs and their annotation can be represented in the tool, to demonstrate how the available language constructs can be used, and to show that the success of automated verification can be influenced by the way the programs and their annotation are modelled.

To this end, we discuss two annotated parallel algorithms from [FvG99]: "A Simple Election Algorithm" and "The Initialization Protocol". In Section 3.2 and Section 3.3 we discuss how the algorithms, which are completely annotated in the style of [FvG99], can be translated to the notation used by the tool and we discuss alternative variants and models of the algorithms.

The annotation of the algorithms has been verified using the tool in combination with the theorem provers PVS and Isabelle. In Section 3.4 the results of the verifications are discussed. The problems encountered are discussed, in combination with possible solutions.

3.2 A Simple Election Algorithm

In this section the modelling of "A Simple Election Algorithm" and its annotation (both from [FvG99, Chapter 24]) is discussed. Every component c in this algorithm has a private boolean variable $y(c)$ and the only thing the component does is performing *one* assignment to this variable. The goal of this algorithm is to synchronise the components in such a way that they all terminate and leave the system in a final state satisfying:

$$\langle\, \#j\colon y(j) \,\rangle \;=\; 1$$

In Section 3.2.1 we show how the original version of [FvG99] can be represented in the tool.

For the correctness of (the annotation of) parallel programs it is important to know which atomic actions are available. [FvG99] introduces the notion of "one-point statement". These are statements of which is assumed that they can be executed atomically. The original algorithm contains a statement that is not one-point. In Section 3.2.2 the transformation of the original algorithm into a finer-grained version that contains only one-point statements is discussed. Even though we know from [FvG99] that this transformation is correctness-preserving, the correctness of this version is also verified using the tool. In Section 3.2.3 the solutions are evaluated.

3.2.1 Original algorithm

Figure 3.1 shows a representation of the original algorithm and its annotation, in the format used by the tool. There are two important differences between our representation of the algorithm and the original in [FvG99].

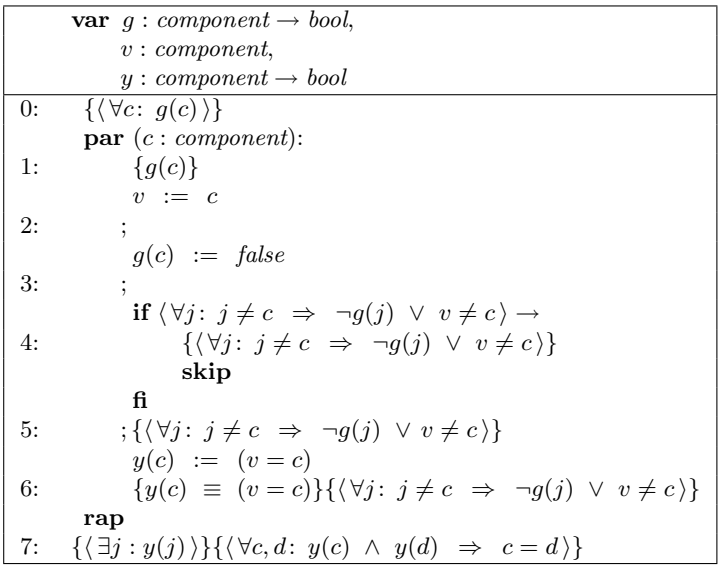

Figure 3.1: Simple Election Algorithm

Slightly different post-condition Theorem provers often define the #-operator recursively, therefore one is almost forced to use induction when proving lemmas involving this operator. Proofs that require induction are difficult to automate. Therefore, we try to avoid it whenever possible. The post-condition of the program is defined using the #-operator, but is logically equivalent to the following condition:

$$\langle\, \exists j\colon\, y(j)\,\rangle \;\wedge\; \langle\, \forall c, d\colon\, y(c) \wedge y(d) \;\Rightarrow\; c = d\,\rangle$$

This condition does not contain the #-operator: induction is avoided by using this as post-condition.

Difference in the interpretation of the if-statement For the design and the verification of parallel programs it is important to know which statements are atomic (indivisible). The interpretation of the if-statement of [FvG99] differs from the interpretation of the if-statement used by the tool. In [FvG99] the evaluation of the guard and the execution of the body are part of the same atomic action. This is not the case when using the tool. For the tool, the evaluation of the guard is one atomic action and the execution of the statements of the body is not part of that same atomic action. If, for the sake of clarity, angular brackets are used to denote the granularity of actions, then the if-statement of [FvG99] can be written as:

$$\{P\}\ \langle\, \textbf{if}\ B \rightarrow S\ \textbf{fi}\,\rangle\ \{Q\}$$

with guard B, statement S, and assertions P and Q. Using the tool, this same fragment can be written as:

$$\{P\}\ \textbf{if}\ \langle\ B\ \rangle \rightarrow \{C\}\ \langle\ S\ \rangle\ \textbf{fi}\ \{Q\}$$

provided the extra condition C is chosen such that:

- $\{C\}\ S\ \{Q\}$ is a correct Hoare-triple,
- $[P \wedge B \Rightarrow C]$, and
- $\{C\}$ is a globally correct assertion.

Unfortunately it is not always possible to rewrite if-statements in the notation used by [FvG99] to this format. Consider, for example, the following program fragment in the notation used by [FvG99]:

$$\langle \textbf{if } a \rightarrow b \ := \ a \textbf{ fi} \rangle \{b\}$$

where a and b are boolean variables. If the transformation above is used to rewrite this fragment we get:

$$\textbf{if } \langle \ a \ \rangle \rightarrow \{C\} \ \langle b \ := \ a \rangle \textbf{ fi } \{b\}$$

and we would have to find a C such that:

- $\{C\}\ b \ := \ a\ \{a\}$ is a correct Hoare-triple,
- $[a \Rightarrow C]$, and
- $\{C\}$ is a globally correct assertion.

To satisfy the first two requirements, we have no other choice but to choose a for C. Consequently, we are presented with an impossible task: in order to satisfy the third requirement we have to prove that assertion $\{a\}$ is globally correct, which is not necessarily true.

Fortunately, in [FvG99] the statement S is almost exclusively a **skip** statement, a so-called guarded **skip**. If statement S is a **skip**, then there always exists a translation to the notation used by the tool: condition Q can be chosen for C. The two notations are then equivalent in the sense that they are both correct, or they are both incorrect.

Recall that atomic statements are modelled in the tool as their *wlp*. It is up to the user to decide how coarse-grained the atomic statements are allowed to be. Instead of using the if-statement of the tool, the user could model if-statements of [FvG99] as atomic statements by defining the *wlp* of the statement:

$$\begin{array}{ll} & [\, wlp.\langle \textbf{if } B \rightarrow \mathrm{S} \textbf{ fi} \rangle.Q \,] \\ \equiv & \\ & [\, B \ \Rightarrow \ wlp.S.Q \,] \end{array}$$

3.2.2 A finer-grained solution

One-point statements are statements that contain at most one reference to at most one shared or private variable. They can be implemented by at most one memory access to at most one shared or private variable and therefore they can be executed atomically. The guard of the guarded statement in Figure 3.1 is not one-point. Next, we are going to transform the original solution to a finer-grained solution with only one-point statements, as described in [FvG99].

[FvG99] mentions the "guard conjunction lemma". This lemma states that if the guard of a guarded **skip** is a conjunction, and one of the two conjuncts is globally correct, then the guarded **skip** can be replaced by two (finer-grained) guarded **skip**s, one for each conjunct, provided the guarded skip with the globally correct guard is executed first.

The conjuncts of the guard of the guarded **skip** are globally correct. As described in [FvG99], the guard conjunction lemma can be applied to replace the guarded **skip** with the following for-statement:

```
for j : j ≠ c do
    if ¬g(j) ∨ v ≠ c → skip fi
od
```

Mohamed G. Gouda's theorem, which states that a disjunction may be evaluated disjunct-wise without impairing the total correctness of the algorithm if one of the disjuncts is globally correct, is also mentioned in [FvG99]. Because disjunct $\neg g(j)$ is globally correct, this theorem can be used to rewrite the if-statement into:

for $j : j \neq c$ **do**
  if $\neg g(j) \rightarrow$ **skip**
  [] $v \neq c \rightarrow$ **skip**
  fi
od

Because the order in which the bodies are executed does not matter, and the body effectively consists of a single atomic statement, the for-statement can be replaced by a parallel composition. In this case, as often is the case, using a parallel construct instead of a loop makes the annotation much clearer and shorter because, for one thing, we do not have to maintain a loop invariant.

The parallel composition is a composition over the elements of type *component*, but the original for-statement ignores element c. Another guarded **skip** is added to the if-statement to ignore this element:

par $(k : component)$:
  if $c = k \rightarrow$ **skip**
  [] $\neg g(k) \rightarrow$ **skip**
  [] $v \neq c \rightarrow$ **skip**
  fi
rap

Note that, if the loop body of the original for-statement would not have been a guarded skip, then the guard of the other two guarded **skip**s would have been strengthened with conjunct $c \neq k$. We have arrived at our goal, a finer-grained solution with only one-point statements, as shown in Figure 3.2.

3.2.3 Evaluation

The tool generates 27 proof obligations for the original solution, and 60 proof obligations are generated for the finer-grained solution. It was to be expected that the number of proof obligations generated for the finer-grained solution is higher, but what is more important is the complexity of the proof obligations. In this case the proof obligations do not become more complex, and the finer-grained solution if preferred.

3.3 The Initialization Protocol

In this section the verification of "The Initialization Protocol" as described in [FvG99, Chapter 19] is discussed. The goal of this algorithm is to distribute the initialisation task of a system over various components, and let the components start on their computation proper only after the total initialisation task has been completed. Solutions for two and three components are discussed in [FvG99].

In Section 3.2 we have seen the parallel composition over the elements of a type. "The Initialization Protocol" consists of two or three separate components which are executed in parallel using a different type of parallel composition: binary parallel composition. This composition is both commutative and associative.

Binary parallel composition was originally not supported by the tool, and we originally had no other choice but to translate the original algorithms to alternative representations without this binary parallel composition. Binary parallel composition can be translated to parallel composition

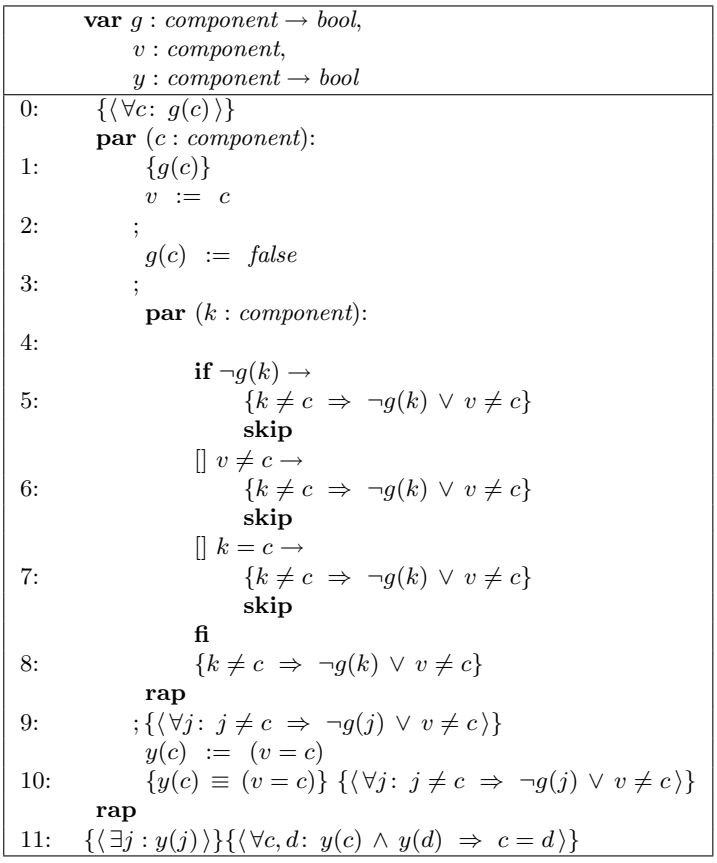

Figure 3.2: Simple Election Algorithm (finer-grained)

over the elements of a type, but the resulting program requires more (admittedly mostly trivial) proof obligations and is more difficult to read and understand. This translation is discussed in Section 3.3.1.

During modelling and verification it became obvious that this translation of the parallel composition is awkward, that it results in a less readable algorithm, and that the resulting algorithm requires a large number of extra proof obligations, even for an algorithm of this size. Fortunately, the developers of the tool agreed that a binary parallel operator would be a valuable addition to the tool, and they added it to the tool.

The solutions for two and three components are discussed in Section 3.3.2 and Section 3.3.3, respectively. For both the two components and the three components solutions we first discuss how the original solutions from [FvG99] can be translated to the notation used by the tool. Then we discuss how the original solutions are translated to alternative representations that use parallel composition over the elements of a type (Variant 1).

The components of the protocol are symmetric. This symmetry can be exploited, which results in a much shorter algorithm with fewer proof obligations, which is much easier to read and understand. We work towards solutions that exploit this symmetry (Variant 2 and Variant 3). They show that parallel composition over the elements of a type also has advantages.

3.3.1 Translate the parallel statement

In this section we show how a parallel program using a binary parallel composition can be translated to a parallel program that uses a parallel composition over the elements of a type.

We consider a parallel program that executes two components in parallel, component X and component Y, with pre-condition P and post-condition Q. The idea is to translate this program to something like:

$\{P\}$
par (c : *component*):
    $\{P\}$
    if $c = X \rightarrow$
        X
    [] $c = Y \rightarrow$
        Y
    fi
rap
$\{Q\}$

where *component* is a type with exactly two elements: X and Y. The reader's first impression may be that this is a correct translation, but unfortunately it is not. To illustrate this we define X as $\{x = 0\}x := x+1$ and we define Y as **skip**. If the pre-condition of the program is $x = 0$, then it is obvious that the assertion in component X is both locally and globally correct. If we apply the (naive) translation above, we get the program:

$\{x = 0\}$
par (c : *comp*):
    $\{x = 0\}$
    if $c = X \rightarrow$
        $\{x = 0\}$
        $x := x + 1$
    [] $c = Y \rightarrow$
        skip
    fi
rap

Recall that assertion $\{x = 0\}$ in a component c is globally correct if, for each atomic action $\{P\}S$ [1] taken from another component c', $\{P \wedge x = 0\}S\{x = 0\}$ is *true*. One of the proof obligations for global correctness of assertion $\{x = 0\}$ for program above would be:

$$\langle \forall c, c' : \; c \neq c' \wedge x = 0 \;\Rightarrow\; x + 1 = 0 \rangle \tag{3.1}$$

which is obviously not valid. The problem is that we are required to prove that assertions are globally correct under statements of their own (original) component. We can solve this problem by adding assertions $\{c = X\}$ and $\{c = Y\}$ to the control points that represent the original control points of component X and component Y, respectively. After adding these assertions we get:

[1] atomic action S with pre-condition P

```
{x = 0}
par (c : comp):
    {x = 0}
    if c = X →
        {c = X}{x = 0}
        x := x + 1
    [] c = Y →
        {c = Y}
        skip
    fi
rap
```

The local correctness of the extra assertions is established by the guards. Global correctness is guaranteed because variable c is not a program variable and cannot be modified. After the change, proof obligation (3.1) is changed into:

$$\langle \forall c, c' :\ c \neq c' \ \wedge\ c = X \ \wedge\ c' = X \ \wedge\ x = 0 \ \Rightarrow\ x + 1 = 0 \rangle$$

which is valid because the antecedent is *false*. This is an application of the rule of disjointness as described in [FvG99].

3.3.2 Two components

In this section we discuss multiple variants of "The Initialization Protocol" for two components. We start with the original algorithm and then work towards a much shorter, fully parameterised version.

Original algorithm The algorithm is shown on the left side of Figure 3.3. There are two components: X and Y. We define a type *component*, the elements of this type represent the components. Because we need two elements, the *component* type is defined as an alias for *bool*, where $X \equiv$ *true* and $Y \equiv$ *false*.

The solution of [FvG99] uses RX and RY to model the termination of the initialisation part of component Y and X, respectively. We introduce variable $r(c)$ for this purpose, where $r(c) \equiv$ *true* if the initialisation part of component c has terminated.

As was the case for the "Simple Election Algorithm", we have an extra control point (c.f. control point with label 5). This is caused by the different interpretation of the if-statement (see Section 3.2.1).

Variant 1: Alternative parallel composition The translation described in Section 3.3.1 is used to translate the original algorithm to a variant that uses a parallel composition over the elements of type *component*. Variant 1 is shown on the right side of Figure 3.3.

When using the tool it is not allowed to put an assertion at the end of the body of an if-statement. Because the original components have a post-condition, a **skip** has to be inserted directly after this post-condition (c.f. control point 8). This does not influence the local or global correctness of the original algorithm.

Variant 2: Parameterised Using parallel composition over the elements of the type *component*, the symmetry of the components can be exploited. In the model the assertions, guards and statements inside the parallel statement have an extra parameter that identifies the component they are evaluated or executed in. This extra parameter can be used to determine the correct statement, assertion or guard for the component, at every point in the program. Because

```
     type component = bool
     const X  ≡  true
           Y  ≡  false
     var c, d : bool,
         x, y : bool,
         r : component → bool
0:   {¬c}{¬d}                                  {¬c}{¬d}
     co                                        par (b : component):
       proc
1:       {¬d}                                      {b ⇒ ¬d}{¬b ⇒ ¬c}
         r(X)  :=  true                            if b →
2:     ;{¬d}{r(X)}                                     {b}{¬d}
         y  :=  false                                  r(b)  :=  true
3:     ;{¬d}{r(X)}{¬y ∨ c}                           ;{b}{¬d}{r(b)}
         x, d  :=  true, true                          y  :=  false
4:     ;{r(X)}{¬y ∨ c}{y ⇒ r(Y)}                     ;{b}{r(b)}{¬y ∨ c}{y ⇒ r(¬b)}
         if y →                                        x, d  :=  true, true
5:           {r(X)}{c}{r(Y)}                         ;{b}{r(b)}{¬y ∨ c}{y ⇒ r(¬b)}
             skip                                      if y →
         fi
6:     ;{r(X)}{c}{r(Y)}                                    {b}{r(b)}{c}{r(¬b)}
         x, d  :=  true, true                              skip
                                                       fi
7:       {c}{r(Y)}{x}                                ;{b}{r(b)}{c}{r(¬b)}
       corp                                            x, d  :=  true, true
       proc
8:       {¬c}                                        ;{b}{c}{r(¬b)}{x}
         r(Y)  :=  true                                skip
                                                     [] ¬b →
9:     ;{¬c}{r(Y)}                                       {¬b}{¬c}
         x  :=  false                                  r(b)  :=  true
10:    ;{¬c}{r(Y)}{¬x ∨ d}{x ⇒ r(X)}                 ;{¬b}{¬c}{r(X)}
         y, c  :=  true, true                          x  :=  false
11:    ;{¬c}{r(Y)}{¬x ∨ d}{x ⇒ r(X)}                 ;{¬b}{r(X)}{¬x ∨ d}{x ⇒ r(¬b)}
         if x →                                        y, c  :=  true, true
12:          {r(Y)}{d}{r(X)}                         ;{¬b}{r(b)}{¬x ∨ d}{x ⇒ r(¬b)}
             skip                                      if x →
         fi
13:    ;{r(Y)}{d}{r(X)}                                    {¬b}{r(b)}{d}{r(¬b)}
         y, c  :=  true, true                              skip
                                                       fi
14:      {d}{r(X)}{y}                                ;{¬b}{r(b)}{d}{r(¬b)}
       corp                                            y, c  :=  true, true
     oc
15:                                                  ;{¬b}{d}{r(¬b)}{y}
                                                       skip
                                                     fi
16:
                                                  rap
```

Figure 3.3: Initialization Protocol for two components. Original left, alternative parallel composition right.

the structure of the two components is identical, we combine the different components, thereby eliminating the if-statement.

The difference between the if-statements in [FvG99] and the if-statement of the tool has already

been discussed in Section 3.2.1. Because of this difference, the original:

$$\{B \Rightarrow C\}\ \langle \textbf{if } B \rightarrow \textbf{skip fi}\rangle\ \{C\}\ S\ \{D\}$$

can be rewritten into:

$$\{B \Rightarrow C\}\ \textbf{if } B \rightarrow \{C\}\ S\ \textbf{fi}\ \{D\}$$

when using the tool. This is used to put the assignments directly following the guarded skip inside the (body of the) if-statement, thereby eliminating a statement (**skip**), a control point, and the assertions on this control point.

This leads to the parameterised version as shown in Figure 3.4. In this figure, the body of the parallel statement consists of two columns, representing the two different components at the different points in the program. An assertion at control point 5 is for example:

$$\lambda b\colon bool.\ \ (b \Rightarrow c)\ \wedge\ (\neg b \Rightarrow d)$$

And the assignment after control point 2:

$$\lambda b\colon bool.\ \ if\ b\ \ then\ \ (y\ :=\ false)\ \ else\ \ (x\ :=\ false)$$

	type $component = bool$ **var** $c, d : bool$, $x, y : bool$, $r : component \rightarrow bool$	
0:	$\{\neg c\}\{\neg d\}$ **par** $(b : component)$:	
1:	**b**: $\{\neg d\}$ $r(b)\ :=\ true$	$\neg$**b**: $\{\neg c\}$ $r(b)\ :=\ true$
2:	$;\{\neg d\}\{r(b)\}$ $y\ :=\ false$	$;\{\neg c\}\{r(b)\}$ $x\ :=\ false$
3:	$;\{\neg y \vee c\}\{r(b)\}\{y \Rightarrow r(\neg b)\}$ $x, d\ :=\ true, true$	$;\{\neg x \vee d\}\{r(b)\}\{x \Rightarrow r(\neg b)\}$ $y, c\ :=\ true, true$
4:	$;\{\neg y \vee c\}\{r(b)\}\{y \Rightarrow r(\neg b)\}$ **if** $y \rightarrow$	$;\{\neg x \vee d\}\{r(b)\}\{x \Rightarrow r(\neg b)\}$ **if** $x \rightarrow$
5:	$\{c\}\{r(b)\}\{r(\neg b)\}$ $x, d\ :=\ true, true$	$\{d\}\{r(b)\}\{r(\neg b)\}$ $y, c\ :=\ true, true$
	fi	**fi**
6:	$\{c\}\{x\}\{r(\neg b)\}$	$\{d\}\{y\}\{r(\neg b)\}$
	rap	

Figure 3.4: Initialization Protocol for two components. Parameterised.

Variant 3: Fully parameterised We can even go one step further than the previous variant. In this version we replace the variables c and d with function $c\colon component \rightarrow bool$, and the variables x and y are replaced with function: $x\colon component \rightarrow bool$. Because the variables have a symmetric role in the components, and every variable has a counterpart in the other component, we can make even more use of the parameter we get from the parallel composition, see Figure 3.5. This way we get rid of the if-then-else constructs and we arrive at a fully parameterised version.

Number of proof obligations The following table shows the number of generated proof obligations for the different versions presented.

```
type component = bool
var c : component → bool,
    x : component → bool,
    r : component → bool
0:   {∀b: ¬c(b)}
     par (b : component):
1:       {¬c(b)}
         r(b)  :=  true
2:       ;{¬c(b)}{r(b)}
         x(¬b)  :=  false
3:       ;{¬(x(¬b)) ∨ (c(¬b))}{r(b)}{x(¬b) ⇒ r(¬b)}
         x(b), c(b)  :=  true, true
4:       ;{¬(x(¬b)) ∨ (c(¬b))}{r(b)}{x(¬b) ⇒ r(¬b)}
         if x(¬b) →
5:           {c(¬b)}{r(b)}{r(¬b)}
             x(b), c(b)  :=  true, true
         fi
6:       {c(¬b)}{x(b)}{r(¬b)}
     rap
```

Figure 3.5: Initialization Protocol for two components. Fully parameterised.

Solution	*Number of proof obligations*
Original	216
Alternative parallel composition (Variant 1)	676
Parameterised (Variant 2)	75
Fully parameterised (Variant 3)	75

The parameterised versions (Variant 2 and Variant 3) are smaller, much clearer, require considerably fewer proof obligations and the proof obligations should not be more difficult to prove. The fully parameterised version is preferred.

3.3.3 Three components

In this section solutions for three components are discussed. The original algorithm from [FvG99] and three variants are discussed. The variants are comparable with the variants used for the solution for two components.

Original algorithm First we translate the algorithm from [FvG99] to the notation used by the tool. This time we cannot get away with making *component* an alias for *bool*: we need a type with three elements. We define *component* to be a so-called enumeration type with the elements: X, Y and Z. Variable $r(c)$ is again used to denote that the initialisation part of component c has terminated. Because of the length of the algorithm, and the symmetry between the components, we show only one component completely in Figure 3.6.

Variant 1: Alternative parallel composition The translation discussed in Section 3.3.1 is used to translate the original algorithm to a variant that uses a parallel composition over the elements of type *component*.

In the original version, the first thing every component does is setting two variables to *false*. The order in which the components execute these two assignments does not matter at all, which we illustrate by executing them in parallel.

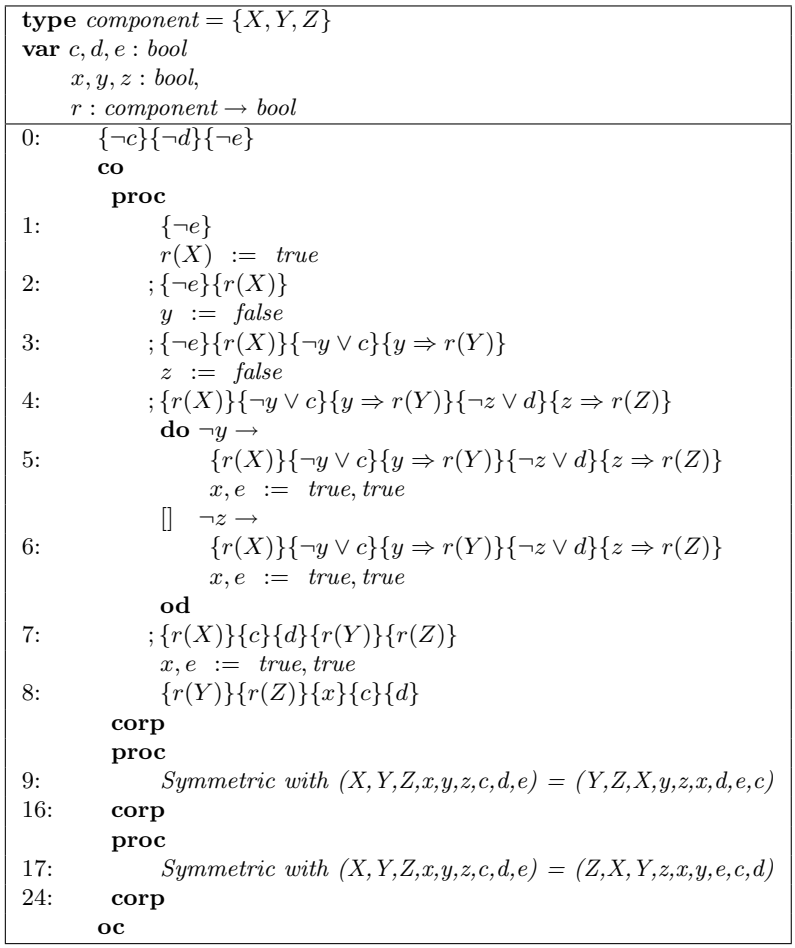

Figure 3.6: Initialization Protocol for three components.

Variant 2 and 3: Parameterised and Fully parameterised Using the symmetry as shown for the solution for two components is more difficult here as there are three components. We have (again) two alternatives:

1. Use functions like

$$\lambda c\text{: } component.\ \ if\ \ (c = X)\quad then\ \ x\ \ \ else\ \ (if\,(c = Y)\ then\ y\ \ else\ z)$$

   This is Variant 2.

2. Replace variables x, y and z with *x: component* $\rightarrow$ *bool*, and replace variables c, d and e with *c: component* $\rightarrow$ *bool* (Variant 3). In the body of the parallel statement we then need functions to refer to the other two components. We name these functions *left* and *right*. Imagine the components form a circle, and that each component has a different component on each side. The *left* and *right* functions give the left and right neighbour, respectively.

Variant 3 results in a much clearer annotation and is a much better way to use and highlight the symmetry. Figure 3.7 shows the result when this variant is used; the implementation of the *left* function and the *right* function is shown in the lower part of the figure.

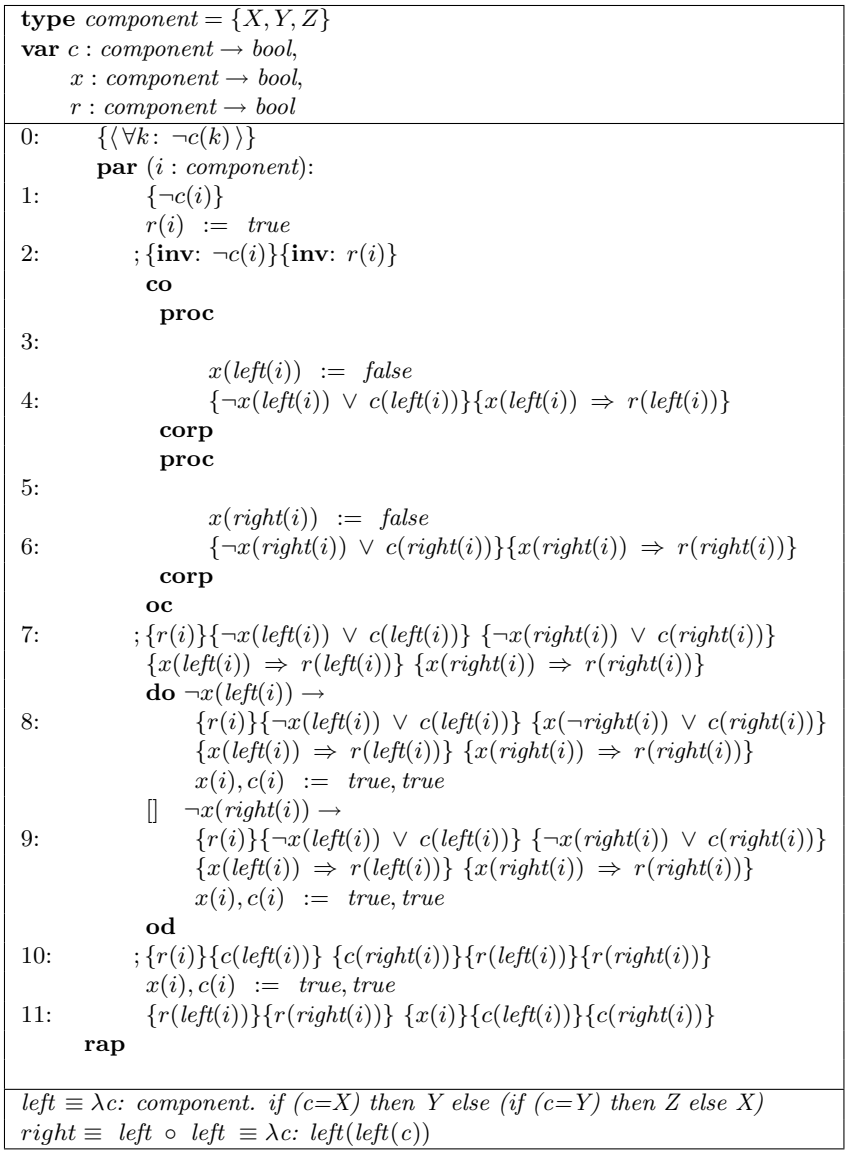

Figure 3.7: Initialization Protocol for three components. Fully parameterised

Number of proof obligations The following table shows the number of proof obligations for the different versions.

Solution	*Number of proof obligations*
Original from [FvG99] (Figure 3.6)	1278
Alternative parallel composition (Variant 1)	2619
Parameterised (Variant 2)	236
Fully parameterised (Variant 3, Figure 3.7)	236

3.4 Verification

In this section we discuss the automated verification of the algorithms mentioned in this report using the theorem provers PVS and Isabelle.

3.4.1 Verification using PVS

In this section the verification of the various algorithms mentioned in this document, using PVS, is discussed. The proof script of [MW05] is used for the automated verification.

A Simple Election Algorithm

The proof obligations of both solutions have been proved automatically, without any manual interaction. Replaying the proofs takes only a few seconds in total.

Initialization Protocol

The original solutions of the "Initialization Protocol" for two and three components and their variants have been verified automatically using PVS. The verification of the solutions for three components revealed some peculiarities of PVS. Only after an addition to the proof script all the proof obligations could be proved automatically, without requiring any interaction of the user. First two additions to the proof script are discussed, followed by the runtime of the automated proofs.

parallel statement One of the proof obligations for the parallel statement is that the post-conditions of the bodies of the parallel statement together imply the post-condition of the parallel statement. For the solutions for three components, some of these could not be proved automatically. This problem occurred only in the initial versions, when binary parallel composition was not yet supported by the tool and a parallel composition over *component* was used for the parallel composition of the two assignments in the variants. The proof obligations that could not be proved automatically are of the form:

$$\frac{\langle\, \forall(c{:}T){:}\ \ P(c) \Rightarrow Q(c) \,\rangle \quad \langle\, \forall(c{:}T){:}\ \ P(c) \,\rangle}{R}$$

where T is the type of the variable used for the parallel composition. PVS fails to conclude that for proof obligations of this form $\langle\, \forall(c{:}T){:}\ \ Q(c) \,\rangle$ holds, which is often required for the proof of R. The tool is changed to instruct PVS to rewrite the proof obligations of this form to:

$$\frac{\langle\, \forall(c{:}T){:}\ \ Q(c) \,\rangle \quad \langle\, \forall(c{:}T){:}\ \ P(c) \,\rangle}{R}$$

This is possible because this part of the structure of the proof obligation is known by the tool. After this addition, the proof obligations of this type can be proved completely automatically.

Type predicates The *component* type for the solutions for three components was originally defined in PVS as

```
component:  TYPE = {X,Y,Z}
```

A few proof obligations of Variant 3, the fully parameterised variant, could not be proved automatically with the original proof script because PVS did not always use the fact that X, Y and Z are the only possible values a variable of type *component* can have. By adding an extra parameter to the `skosimp*` tactic ([SORSC01, Section 4.14.1]) we instruct PVS to automatically add constraint information of introduced skolem variables to the antecedents. This does not work when using the original definition of *component*, because it is not a predicate subtype. We change the definition of *component* to a predicate subtype:

```
X: nat = 0
Y: nat = 1
Z: nat = 2
component: type = {x:nat | x=X OR x=Y OR x=Z}
```

Using this definition in combination with the extra argument for `skosimp*`, PVS generates an assumption with the following type information

$$i = X \ \vee \ i = Y \ \vee \ i = Z$$

for every introduced skolem variable i of type *component*.

Using the new proof script, and the new definition of component, PVS is able to prove all the proof obligations completely automatically.

Runtime The following table gives an indication of the time it takes to replay the proof[2]:

Algorithm	*Proof obligations*	*Runtime (sec)*
2 components		
Original	216	12
Variant 1: Alternative parallel composition	676	73
Variant 2: Parameterised	75	7
Variant 3: Fully parameterised	75	7
3 components		
Original	1278	118
Variant 1: Alternative parallel composition	2619	294
Variant 2: Parameterised	236	75
Variant 3: Fully parameterised	236	188

The time it takes to prove the proof obligations is more or less proportional to the number of proof obligations for all but the third variant of the algorithm for three components. The reason the verification of the third variant takes more than twice as long as the verification of the second variant is that case distinction on the components is required for the verification of the third variant. This is caused by the extra type information that is added to the assumptions by *skosimp*. By changing the definition of *left* and *right*, we eliminate the need for these extra assumptions and eliminate the need for the case distinctions. We discuss this slightly different model.

[2] An Intel Xeon 2.8GHz processor is used. Only a small fragment of the 4GB memory available is actually used.

The functions *left* and *right* are defined in PVS as follows:

```
left(c: component): component =
    IF c = X THEN Y
              ELSE IF c = Y THEN Z
                            ELSE X
              ENDIF
    ENDIF

right: [component -> component] =
    (left o left)
```

Only two properties of *left* and *right* are actually needed for the proofs: the original specification of right, and one property of left. We use an alternative specification of the *left* function, we specify only the type and the required property of *left*:

```
    left: { f: [component -> component] |
                FORALL (c,d: component): c = d OR c = f(d) OR c = (f o f)(d) }
```

left now represents an arbitrary function satisfying the property:

$$\langle\, \forall c, d\colon\ c = d \ \vee\ c = \mathit{left}(d) \ \vee\ c = \mathit{right}(d) \,\rangle$$

PVS generates a TCC (Type Correctness Condition) for this definition of *left*, which forces us to prove that there exists at least one function instance of type *left*, that satisfies the correct property. This can be proved by supplying a witness: the original definition of *left*.

Unfortunately the proof script does not use the type predicates for functions and variables that are not skolem variables, and thus the property of left is not always used in the proofs. We need a system invariant to explicitly state the property of *left*.

Using this alternative specification, and the extra system invariant, we don't need the type predicates of *component* anymore, and we can go back to the original definition of *component*: `component: TYPE = {X, Y, Z}`.

Using the new specification of *left*, the new system invariant, and the original definition of *component*, the proof script does not apply case analysis on skolem variables of type *component* anymore, and the proof is much faster; the runtime of the verification is 71 seconds.

Although the verification turns out to be much faster when this slightly modified model is used, using something similar for future projects is only advisable if the required properties are known beforehand, if they are obvious. For this algorithm, it may be more than 100 seconds faster, but it took a lot more time and effort to determine which properties are required. Our goal is not as much to get the best possible runtime of the proof, but is to reduce the human effort.

3.4.2 Verification using Isabelle

In this section we focus on the influence the models of the algorithms (and their annotation) have on the automated verification with Isabelle. We use *Proof script A*, which is described in Section 4.4.2, for this automated verification. More detailed information about Isabelle and the automated verification using Isabelle can be found in Chapter 4.

A Simple Election Algorithm

The proof obligations of both solutions have been proved automatically, without any manual interaction. Replaying the proofs takes only a few seconds in total.

Initialization Protocol

Initially not all of the parameterised versions could be verified completely automatically, but after a few changes to the model of the algorithm in Isabelle, all proof obligations can be proved fully automatically using the default proof script. First the changes to the model are discussed, and then an indication of the time it takes Isabelle to replay the proofs is given.

Component type We have defined the *component* type for the solutions for three components in Isabelle as

```
datatype component = X | Y | Z
```

The proof of some proof obligations for the parameterised variants need the fact that X, Y and Z are the only possible values for a variable of type *component*. When defining the datatype one of the theorems Isabelle automatically defines and proves is:

$$\langle \forall (c\colon component) :\ \ c = X \ \vee\ c = Y \ \vee\ c = Z \rangle$$

But, the tactics of Isabelle do not automatically use this fact, and some proof obligations cannot be proved automatically. After adding this theorem as a new system invariant, the problematic lemmas can be proved automatically.

We have encountered the same problem with the verification of the proof obligations with PVS. In PVS we were able to solve this problem by using type predicates, but type predicates are not available in Isabelle. We have shown how an alternative PVS model can be used to reduce the runtime. This option is not available in Isabelle because we are not able to prove all proof obligations automatically in Isabelle, using only the one property of *left*, and not using the system invariant above.

if-then-else construct For the parameterised variant (Variant 2) of the solution for two components, the *component* type is just an alias for *bool*, and if-then-else constructs are used to select the assertion matching the component. Several proof obligations using if-then-else constructs could be proved automatically with Isabelle. The problem is that the if-then-else construct is not always simplified when it appears in the assumptions of a goal. We have to add (and prove) a new rule that states that

$$(\mathit{if}\ b\ \mathit{then}\ c\ \mathit{else}\ d) \quad \equiv \quad ((b \wedge c)\ \vee\ (\neg b \wedge d))$$

and instruct Isabelle to apply this rule automatically to rewrite the left hand side to the right hand side. After this change all the lemmas can be proved automatically.

Runtime The following table gives an indication of the time it takes to replay the proof[3]:

Algorithm	*Proof obligations*	*Runtime (sec)*
2 components		
Original	216	18
Variant 1: Alternative parallel composition	676	124
Variant 2: Parameterised	75	12
Variant 3: Fully parameterised	75	9
3 components		
Original	1278	321
Variant 1: Alternative parallel composition	2619	2143
Variant 2: Parameterised	236	154
Variant 3: Fully parameterised	236	118

[3] An Intel Xeon 2.8GHz processor is used. Only a small fragment of the 4GB memory available is actually used.

Isabelle is slower than PVS for the verification of all algorithms but the last variant of the solution for three components. The verification of this algorithm was originally faster in Isabelle, but after the changes to the model of the algorithm in PVS, as discussed in the end of the previous section, the verification in PVS is also faster for this variant.

3.5 Conclusions

We have demonstrated how annotated parallel programs can be modelled in the tool, and we have discussed the available language constructs. We have discussed different models and the effect they have on the automated verification. We have discussed advantages of the two types of parallel composition, and we have shown that binary parallel composition is a valuable addition to the tool. Parallel composition over elements of a type has its advantages; we have shown how it can be used to exploit symmetry between components, to create much shorter and more readable programs.

Chapter 4

Automated verification using Isabelle

4.1 Introduction

In this chapter the automated verification using Isabelle is discussed. A short introduction to Isabelle is given in Section 4.2. We use the Proof Generator tool (see Chapter 2) for our verification with Isabelle. Originally this tool only used PVS, but we have added support for Isabelle as back-end of the tool. In Section 4.3 the Isabelle modelling of programs and annotation is discussed. It is important to have an effective proof strategy; the proof scripts we have developed for Isabelle are discussed in Section 4.4. The changes to the actual code of the tool are discussed in Section 4.5. In Section 4.6 we discuss the results of our experiments with the automated verification of several algorithms using our proof scripts. In Section 4.7 the problematic proofs are evaluated, and possible solutions to the problems encountered are discussed.

4.2 Introduction to Isabelle

Isabelle [Pau94] is a theorem prover developed at Cambridge University and the TU Munich. Isabelle is generic, different logics are available and users can add their own logic and tactics. In this report we consider Isabelle 2004 in combination with the logic HOL, the Higher Order Logic. When we refer to Isabelle, we are actually referring to the combination Isabelle/HOL. This section is an introduction to the aspects of Isabelle that are relevant for this report.

The specification language of Isabelle is similar to functional programming languages like ML and Haskell. Functions are curried, they are lambda abstractions and they are total, i.e. they are defined for all possible input values.

Types Every term in Isabelle is typed. Types in Isabelle are non-empty. Function types in Isabelle are denoted by the function type operator $\Rightarrow$. Function application is left associative and the function type operator is right associative.

If M is a term of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ (notation: $M :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$) and N is a term of type τ_1 then the application of M to N (notation: $M\ N$) is a term of type $\tau_2 \Rightarrow \tau_3$. The term $f\ x\ y$ should be read as $(f\ x)\ y$, the application of $f\ x$ to y.

Isabelle uses a system similar to Haskell's to determine ("infer") the type of a term using the context. One of the advantage of this is that we do not need to specify the type of every term explicitly.

Representation of goals Isabelle uses a meta-logic to represent theorems and goals. This meta-logic is independent of the actual logic used (HOL in our case). Generalised Horn clauses are used to represent the goals and theorems in this meta-logic; they are of the following form:

$$[\![\phi_1 ; \ldots ; \phi_n]\!] \quad \Longrightarrow \quad \psi$$

where $0 \leq n$. This states that under the assumption that the conjunction of $\phi_1 \ldots \phi_n$ (the assumptions or premises) holds, we have to prove that ψ (the conclusion) holds. If the sequence of assumptions is empty, then both the brackets and the arrow are omitted. The brackets are also omitted if there is only a single assumption. This goal is logically equivalent to:

$$\langle\, \forall i :\ 1 \leq i \leq n \Rightarrow \phi_i \,\rangle \quad \Rightarrow \quad \psi$$

Variables Isabelle has three kinds of variables. In addition to the usual free and bound variables, a third kind of variable is used: schematic or unknown variables. Schematic variables are defined at the meta-logic, they are placeholders for terms and they can be substituted by a term of the correct type during the proof process. Consider for example the following goal:

$$f\ ?x \Longrightarrow f\ 15$$

The names of schematic variables start with a question mark: *?x* is a schematic variable. The goal can be proved by substituting the term *15* for *?x*. Not every term with the correct type can be substituted for a schematic variable: terms that depend on bound variables cannot be substituted. In Section 6.3.5 an example is discussed that illustrates why this is not permitted.

Proof process Theorem provers are used to prove lemmas or theorems. At any point in the proof process the proof state consists of zero or more subgoals. The initial proof state has one subgoal, the lemma that has to be proved. Tactics, or theorem proving functions, are used to transform a proof state into another proof state: they can change subgoals into subgoals that are logically equivalent or stronger, split them into multiple (smaller) subgoals, or discharge them when they have been proved to be valid. Subgoals are valid when it is trivial they are equivalent to *true*.

To keep proofs sound, tactics are built on top of a small kernel that offers only a very limited number of functions to manipulate the proof state. The following categories of available tactics are of our interest:

- **Unification**

  Unification is used by most tactics. It is used to find instantiations for schematic variables. The *assumption* tactic, for example, attempts to unify the conclusion with the assumptions, in an attempt to find instantiations for schematic variables to prove the goal. Two terms t and u are unifiable if there exists a substitution σ such that $t\sigma = u\sigma$, where the application of substitution to a term is written as a suffix.

  Consider for example the term $f\ a\ ?x$ and the term $f\ ?y\ b$. The two terms are unifiable because with $\sigma = \{?x \mapsto b,\ ?y \mapsto a\}$, both $(f\ a\ ?x)\sigma$ and $(f\ ?y\ b)\sigma$ are equal to $f\ a\ b$. Unification in Isabelle is more complicated than this example as it has to take types into account, higher order unification is used and there are restrictions to the terms the schematic variables can be instantiated with. See [NPW02, Section 5.8] for more information about unification in Isabelle.

- **Natural deduction**

  Natural deduction forms the basis for reasoning in Isabelle. Natural deduction in Isabelle involves unifying a goal with an inference rule, and replacing the goal with one or more goals based on the rule. Such an inference rule in Isabelle is in essence nothing more than a previously proved theorem in the right form.

Natural deduction rules are categorised according to how they are most likely to be used. The categories are: *introduction*, *elimination*, and *destruction*. The standard introduction and elimination rules for the logical connectives and quantifiers as we know them from natural deduction calculus are available. New rules can be defined, and added to the set of default rules, provided they have been proved first. Next, examples are used to demonstrate the three categories of rules.

An example of an **introduction** rule is $\dfrac{?P \quad ?Q}{?P \wedge ?Q}\ (\text{conjI})$

This rule states (justifies) that if we have proved $?P$, and we have proved $?Q$, that we then can conclude (infer) $?P \wedge ?Q$. In Isabelle, this rule is written as:

$$\llbracket\, ?P\,;\ ?Q\,\rrbracket \quad \Longrightarrow \quad ?P \wedge ?Q \tag{conjI}$$

The *rule* tactic applies introduction rules such as *conjI* to a goal. The rules are applied backwards. If the tactic `apply(rule conjI)` is applied to the goal

$$\llbracket\, \phi_1\,;\ \ldots;\ \phi_n\,\rrbracket \quad \Longrightarrow \quad \psi_1 \ \wedge\ \psi_2$$

then the conclusion of the goal is unified with the conclusion of the rule. They unify with the mapping: $\sigma = \{?P \mapsto \psi_1,\ ?Q \mapsto \psi_2\}$. This mapping is applied to the premises of the rule (i.e. *conjI*), and it can be concluded that $\psi_1 \wedge \psi_2$ can be proved by proving that both $(?P)\sigma$ and $(?Q)\sigma$ hold (ψ_1 and ψ_2).

The goal is replaced by the following two subgoals:

- $\llbracket\, \phi_1\,;\ \ldots;\ \phi_n\,\rrbracket \quad \Longrightarrow \quad \psi_1$
- $\llbracket\, \phi_1\,;\ \ldots;\ \phi_n\,\rrbracket \quad \Longrightarrow \quad \psi_2$

An example of an **elimination** rule is:

$$\frac{\begin{array}{ccc} & [?P] & [?Q] \\ & \vdots & \vdots \\ ?P \vee ?Q & ?R & ?R \end{array}}{?R}\ (\text{disjE})$$

In Isabelle, this rule is written as:

$$\llbracket\, ?P \vee ?Q\,;\ ?P \Longrightarrow ?R\,;\ ?Q \Longrightarrow ?R\,\rrbracket \quad \Longrightarrow \quad ?R \tag{disjE}$$

This elimination rule should be read as:

> If we have to prove *?R* and we have already established that $?P \vee ?Q$ holds, then it suffices to show that $?R$ can be proved using the extra assumption $?P$, and that $?R$ can be proved using the extra assumption $?Q$.

The *erule* tactic applies elimination rules to a goal. If `apply(erule disjE)` is applied to the goal

$$\llbracket\, \delta_1 \vee \delta_2\,;\ \phi_1\,;\ \ldots\ ;\ \phi_n\,\rrbracket \quad \Longrightarrow \quad \psi$$

then the *first* premise of the rule is unified with one of the assumptions of the goal, and the conclusion of the rule is unified with the conclusion of the goal. Applied to our example, the first assumption is used as it matches the first premise of the rule. This gives the mapping: $\sigma = \{?P \mapsto \delta_1,\ ?Q \mapsto \delta_2,\ ?R \mapsto \psi\}$. The substitution σ is applied to the elimination rule, and from the theorem that results from the substitution it can be concluded that it suffices to prove $\delta_1 \Longrightarrow \psi$ and $\delta_2 \Longrightarrow \psi$:

- $\llbracket\, \phi_1\,;\ \ldots\ ;\ \phi_n\,\rrbracket \quad \Longrightarrow \quad \delta_1 \quad \Longrightarrow \quad \psi$
- $\llbracket\, \phi_1\,;\ \ldots\ ;\ \phi_n\,\rrbracket \quad \Longrightarrow \quad \delta_2 \quad \Longrightarrow \quad \psi$

These goals are the same as:

- $[\![\phi_1 ; \ldots ; \phi_n ; \delta_1]\!] \implies \psi$
- $[\![\phi_1 ; \ldots ; \phi_n ; \delta_2]\!] \implies \psi$

since $[\![P ; Q]\!] \implies R$ is nothing more than a shorthand notation for $P \implies Q \implies R$.

Destruction rules are rules that transform one of the assumptions of a goal into a weaker assumption. A relevant example of a *destruction* rule is:

$$\text{ALL } x.\ ?P\ x \quad \implies \quad ?P\ ?x \qquad \text{(spec)}$$

This destruction rule is applied using `apply(drule spec)`. The *spec* destruction rule justifies the transformation of an assumption matching $\forall x.\ ?P\ x$ into the weaker assumption $?P\ ?x$.

When a rule is applied, it is possible to supply substitutions for the schematic variables of the rule. For example `apply(drule_tac x="x1" in spec)` applies the destruction rule *spec*, and substitutes $x1$ for the schematic variable $?x$ in the *spec* rule. The effect of this application is that a universal quantifier in the assumptions is instantiated with the term $x1$. If an elimination rule or a destruction rule can be matched with more than one assumption, then the first match in the list of assumption is used. Visually, this is the one most to the left. If the first match in the list is not the assumption that should be used, the *rotate_tac* tactic can be used to change the position of the assumptions in the list. For more information about inference rules in Isabelle, see [NPW02, Section 5].

- **Simplification**

  The *simp* tactic uses conditional rewrite rules of the form

  $$[\![\phi_1 ; \ldots ; \phi_n]\!] \quad \implies \quad (lhs = rhs)$$

  This rule states that if the premises $\phi_1 \ldots \phi_n$ hold in the context of a goal, then *lhs* is equivalent to *rhs* in the goal. This rule justifies rewriting *lhs* to *rhs* in the current goal if the premises of the rule are valid for the current goal. In addition to the large number of simplification rules standardly available in Isabelle, the user can add new rewrite rules to the set of default rewrite rules, and can choose which ones to apply, or not to apply, each time the simplifier is invoked. Not every theorem of the correct form is suitable as rewrite rule. Namely, *rhs* should be smaller than *lhs*, according to some well-founded order, to prevent endless loops; especially for rules that are applied by default. This is not enforced by Isabelle, it is possible for the user to create a rewrite system that doesn't terminate. The *simp* tactic also creates rewrites rules from the assumptions. Apart from rewriting, the *simp* tactic can also solve simple arithmetic expressions. For more information about simplification and rewriting in Isabelle, see [NPW02, Section 2.5.1].

- **Classical reasoning** Classical reasoning uses a set of inference rules (natural deduction rules) and backtracking to search for a proof. Different tactics are available that use classical reasoning. The following tactics in this category are of interest:
  - *safe* and *clarify*. The *safe* tactic applies all safe rules. These are rules that cannot render a goal unprovable. When safe rules are applied to a goal, the goal is transformed into a logically equivalent goal. Rules that can render a goal unprovable are unsafe. The difference between the tactics *safe* and *clarify* is that *clarify* is restricted to safe reasoning steps that do not split the goal into subgoals.
  - *auto* and *force* combine classical reasoning with simplification. While *auto* is applied to all subgoals, *force* is applied to only a single goal, the first subgoal by default. The *force* tactic "tries harder" to prove the goal and therefore it can take longer to terminate [NPW02, Section 5.13]. *force* either proves the goal, or fails if it is not able to prove the goal. *auto* tries to prove as much as possible, and only fails if it did not make any changes. *auto* terminates in a proof state where for each of the subgoals that could not be proved completely, only safe rules and simplification have been applied.

– *blast*, *fast* and *best*. *blast* uses a built-in first order reasoner. *fast* and *best* use standard Isabelle inference. The tutorial [NPW02, Section 5.13] describes the *fast* tactic and the *best* tactic as "legacy methods that work well with rules involving unusual features." For more information about classical reasoning and the tableau method used by *blast* we refer to [Pau99].

- **Arithmetic** The *arith* tactic uses linear algebra to prove a goal.

- **Case distinction** The *case* tactic implements case distinction. If `apply(case_tac "P x")` is applied to the goal

$$[\![\phi_1 ; \ldots ; \phi_n]\!] \quad \Longrightarrow \quad \psi$$

then the goal is split into the following two subgoals:

$$\begin{aligned} [\![\phi_1 ; \ldots ; \phi_n ; P\ x]\!] \quad &\Longrightarrow \quad \psi \\ [\![\phi_1 ; \ldots ; \phi_n ; \neg P\ x]\!] \quad &\Longrightarrow \quad \psi \end{aligned}$$

The first assumes that $P\ x$ holds, the second assumes that $P\ x$ does not hold.

- **Use other theorems** The *insert* tactic adds axioms and theorems to the assumptions of a goal. Theorems can only be added if they have been proved first.

  The *subgoal* tactic introduces assumptions to a goal. If `apply(subgoal_tac "P x")` is applied to the goal

$$[\![\phi_1 ; \ldots ; \phi_n]\!] \quad \Longrightarrow \quad \psi$$

then the goal is split into the following two subgoals:

$$\begin{aligned} [\![\phi_1 ; \ldots ; \phi_n ; P\ x]\!] \quad &\Longrightarrow \quad \psi \\ [\![\phi_1 ; \ldots ; \phi_n]\!] \quad &\Longrightarrow \quad P\ x \end{aligned}$$

The first subgoal is the original subgoal with the extra assumption added and the second subgoal forces the user to prove that the assumption is actually valid.

The *subgoal* tactic is very similar to the *case* tactic. The difference is that the second subgoal is weaker when case distinction is used.

Tactics are applied to the proof state using the *apply* or the *by* function. The difference between the two is that the *by* function not only applies the tactic, but also tries to finish the proof, using the *assumption* tactic and backtracking if needed. The application of a tactic fails, unless it makes changes to the proof state. Tactics that are applied to a single subgoal, are applied to the first subgoal by default. A proof is considered to be finished (lemma is proved) if the original proof state is transformed into a proof state without any subgoals.

Tacticals can be used to combine tactics and to provide control structures. The following tacticals are of interest:

Repetition: T+ Repeat tactic T for as long as T is successful, but at least once. If the first application of T fails, then $T+$ fails.

Optional: T? The application of tactic T is optional. T is applied and $T?$ is successful, even if the application failed.

Choice: T|U T is applied. Only if this application fails, then is tactic U applied.

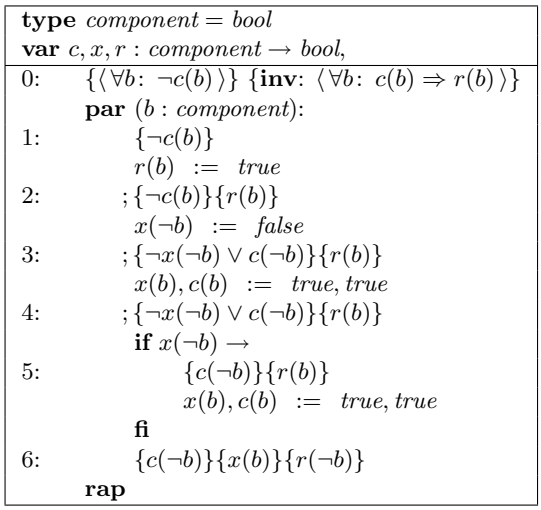

Figure 4.1: Initialization Protocol for two components.

4.3 Modelling of programs and proof obligations

In this section we focus on the Isabelle representation of programs and proof obligations. A variant of "The initialization protocol" for two components (Figure 4.1, discussed in Section 3.3) is used as an example to show how the various aspects can be modelled in Isabelle.

The tool accepts an input file that defines the structure of the annotated program, and contains the name of the file containing the Isabelle model of the statements, guards and assertions. The structure of the program is completely independent of the theorem prover used, and is not modelled in Isabelle.

The file that defines the guards, statements and assertions is called the import file. This file contains the part of the model that is specific for the theorem prover, in this case Isabelle. This import file is discussed first (Section 4.3.1).

The structure of the program, as defined in the input file, is used by the tool to construct the proof obligations, in this case for Isabelle, using the guards, statements and assertions defined in the import file. The proof obligations are part of the output of the tool. The modelling of the proof obligations in Isabelle is discussed in Section 4.3.2.

4.3.1 Model of the program

The user is responsible for modelling the types, guards, assertions and the atomic statements in Isabelle. The modelling of the example program is discussed in this section.

Data types In addition to the elementary types like *bool* and *nat*, which are standardly available in Isabelle, extra types are needed. The example program uses a type *component* to identify processes. For our example there are exactly two components and we define *component* to be an alias for the type *bool*.

```
types
    component = "bool"
```

The data type *state* is used by the tool to denote a state of the program. We define the *state* type as a record type, with a field for each variable of the program.

```
record state =
    c :: "component ⇒ bool"
    x :: "component ⇒ bool"
    r :: "component ⇒ bool"
```

For every field of the record, Isabelle creates a function which, given a *state* instance, delivers the value of the field. For example the function $c :: state \Rightarrow component \Rightarrow bool$ is introduced for the state record type. This function takes an instance of a *state*, and delivers the value of the c field of the *state* instance, which represents the variable c of the program.

```
types
    predicate = "state ⇒ bool"
```

Because a predicate over *state* (*state* ⇒ *bool*) is used frequently, a short hand notation (*predicate*) is created for this type. This alias is also used by the proof obligations generated by the tool. Internally, Isabelle automatically translates the type *predicate* to its definition above.

Annotation Assertions and invariants are modelled as predicates over a *state*.

```
constdefs
    ass_0a :: "predicate"
    "ass_0a  ==  %s.  ALL b.  ~c s b"

    inv_0a :: "predicate"
    "inv_0a  ==  %s.  ALL b.  c s b  -->  r s b"

    ass_3a :: "component ⇒ predicate"
    "ass_3a  ==  %b s.  ~ x s (~ b)  |  c s (~ b)"
```

Functions have a type (const) and a definition (def). The symbol % is used for λ-abstraction and is followed by bound variables. In this case the type of the bound variables can be omitted because they are determined (uniquely) from the environment. Bound variable b in *ass_3a* represents the component identifier and bound variable s represents the state.

Atomic actions Guards are modelled as predicates over a *state*. Atomic statements are modelled as their *wlp*, their *weakest liberal precondition*.

```
constdefs
    guard_4a :: "component ⇒ predicate"
    "guard_4a  ==  %b s.  x s (~ b)"

    wlp_stat_1 :: "component ⇒ predicate ⇒ predicate"
    "wlp_stat_1  ==  %b p s.  p  ( s(| r := (r s)(b := True) |) )"
```

Recall that proof obligations are of the form $[\, P \Rightarrow wlp.S.Q \,]$. Given a state s, a component b and a post-assertion p, *wlp_stat_1* defines $wlp.\langle r(b) := true \rangle.p$. In *wlp_stat_1* field r of the state record is updated. $(r\ s)(b := True)$ is a function update, which results in the function

$$\lambda\, d.\quad if\quad (d = b)\quad then\quad True\quad else\quad (r\ s\ d)$$

This function represents variable r after the assignment $r(b) := true$ at control point 1.

Unfolding definitions Definitions in Isabelle are not automatically unfolded. To automatically unfold our definitions in the proof, the definitions are added as default simplification (rewrite) rules.

```
declare
    inv_0a_def      [simp]
    ass_3a_def      [simp]
    guard_4a_def    [simp]
    wlp_stat_1_def  [simp]
```

It is the user's responsibility to add the relevant definitions to the set of default simplifier rules. In theory it is possible to let the tool add them automatically, and this is usually preferred. But there may be situations where you know beforehand that it is better not to unfold certain definitions. There is a trade-off between convenience and flexibility. We value flexibility and have decided to leave it to the user to decide which definitions to unfold automatically.

4.3.2 Modelling of the proof obligations

In this section the modelling of the proof obligations, which is part of the output of the tool, is discussed. More information about the various aspects that have to be modelled can be found in Section 2.3.

Control points The type of the *lab* variables is defined. Axioms are used to specify which assertions and invariants are located at which control points.

```
consts
    lab_3 :: "bool ⇒ predicate"

axioms
    lab_3_inv_0a :  "ALL (s :: state)(b :: component).  lab_3 b s ⟶ inv_0a s"

    lab_3_ass_3a :  "ALL (s :: state)(b :: component).  lab_3 b s ⟶ ass_3a b s"
```

Scope lemmas *scp* variables and the scope lemmas that define the relation between *scp* variables and invariants are defined.

```
consts
    scp_0 :: "predicate"

axioms
    def_scp_0 : "
      ALL (s :: state).
        scp_0 s  =
              (    lab_0 s
                 | (EX (b :: component).  lab_1 b s)
                   ...
                 | (EX (b :: component).  lab_6 b s)
              )
    "

lemma scp_0_inv_0a: "
    ALL (s :: state).  scp_0 s ⟶ inv_0a s"
```

Proof obligations There are different types of proof obligations, each has a slightly different format. We give an example of a proof obligation for local correctness, and we give an example of a proof obligation for global correctness.

```
lemma loc_ass_3a_stat_2: "
    ALL (s :: state).
      ALL (b :: component).
        lab_2 b s  ⟶
          wlp_stat_2 b (ass_3a b) s"
```

```
lemma glob_ass_3a_stat_1: "
  ALL (s :: state).
    ALL (b :: component).
      ALL (b0 :: component).
        (lab_3 b0 s  &  lab_1 b s) ⟶
          (b ∼= b0)  ⟶
            wlp_stat_1 b (ass_3a b0) s"
```

4.4 Proof strategy

It is vital to have a generic proof script that is able to prove as many proof obligations as possible, with a minimum amount of manual interaction. We distinguish two types of lemmas: scope lemmas and proof obligations. First, the proof strategy for the scope lemmas is discussed in Section 4.4.1. Finally, the proof strategy for the proof obligations is discussed in Section 4.4.2.

4.4.1 Scope lemmas

Scope lemmas are of the form:

$$\text{ALL } (s :: \mathit{state}).\quad \mathit{scp\_0}\ s \ \longrightarrow\ \mathit{inv\_0a}\ s \qquad \text{(scp\_0\_inv\_0a)}$$

scp variables equal a disjunction of *lab* variables (the labels within the scope of the parallel statement), this is defined using an axiom (c.f. *def_scp_0*). For each of the *lab* variables there is an axiom (c.f. *lab_3_inv_0a*) that states that the invariant (c.f. *inv_0a*) is located at a control point (c.f. control point 3). Scope lemmas define the relation between *scp* variables and invariants, and because of the definition of the scope variables (c.f. *scp_0*), they are valid by construction.

```
lemma scp_0_inv_0a: "
  ALL (s :: state).  scp_0 s  ⟶  inv_0a s"
```

```
apply(clarify)
apply(insert def_scp_0, rotate_tac -1)
apply(erule_tac x = "s" in allE, rotate_tac -1)
apply(safe)
apply(tactic {* cut_facts_tac [thm "lab_0_inv_0a"] 1 *}, simp only:)
apply(tactic {* cut_facts_tac [thm "lab_1_inv_0a"] 1 *}, simp only:)
...
apply(tactic {* cut_facts_tac [thm "lab_6_inv_0a"] 1 *}, simp only:)
done
```

Here we see the proof obligation and proof script of scope lemma *scp_0_inv_0a*. The outline of the proof of a scope lemma is as follows: first the universal quantifier and the implication are eliminated, using the *clarify* tactic.

Then the axiom defining the scope lemma is introduced (*insert*) and the universal quantifier of the axiom is instantiated (*erule_tac*). The *rotate_tac* moves the introduced assumption to the head of the list to ensure that the *erule_tac* is applied to the right assumption. The *safe* tactic follows, this tactic splits the goal into subgoals – one for each disjunction in the premises.

For each subgoal the axiom that defines the relation between the label and the invariant is introduced using *cut_facts_tac*. Usually the *insert* tactic is used to strengthen the assumptions with axioms, but here we use the lower level ML function *cut_facts_tac* instead. The reason for this is that the *insert* tactic adds the axiom to the assumptions of every subgoal, while the *cut_facts_tac* function gives us the choice to add axioms to the assumptions of a single subgoal. Adding the axioms to the assumptions of each of the subgoals makes the proof unnecessary complicated, and the time to prove the lemma increases dramatically if there are several invariants.

Finally, the *simp* tactic proves all the subgoals. The *only:* part of the simplification tactic instructs the tactic to use only the simplification rules that follow, in this case none. In this way, only the rewrites rules derived from the assumptions and conclusion are used. This ensures that the definitions are not unfolded, which is not required in this case, and would make the proof more complicated.

4.4.2 Proof obligations

In this section the proof strategy for the proof obligations is discussed. We have experimented with various proof scripts. First we discuss the most successful proof scripts: "Proof script A" and "Proof script B". Finally, various alternatives to the two proof scripts are discussed.

Proof script A

First *Proof script A* is given, and then the proof script is discussed. Afterwards, we discuss the most important decisions using examples.

```
lemma loc_ass_3a_stat_2: "
  ALL (s :: state).
    ALL (b :: component).
      lab_2 b s  ⟶
        wlp_stat_2 b (ass_3a b) s"
```

```
apply(clarify)?
apply(insert lab_2_inv_0a, rotate_tac -1)?
apply(erule_tac x = "s" in allE, rotate_tac -1)?
apply(erule_tac x = "b" in allE, rotate_tac -1)?
apply(insert lab_2_ass_2a, rotate_tac -1)?
apply(erule_tac x = "s" in allE, rotate_tac -1)?
apply(erule_tac x = "b" in allE, rotate_tac -1)?
apply(insert lab_2_ass_2b, rotate_tac -1)?
apply(erule_tac x = "s" in allE, rotate_tac -1)?
apply(erule_tac x = "b" in allE, rotate_tac -1)?
apply(clarsimp)?
apply(safe)?
by((auto | rule exI)+)?
```

The first part of the proof script is similar to the proof script for scope lemmas: *clarify* is used to perform all the obvious reasoning steps, the universal quantifiers and the implications are eliminated from the conclusion, and the appropriate axioms are introduced and their universal quantifiers are instantiated with constants that represent the states and components. The `?` tactical indicates that the tactic is optional. Without this tactical the application of the tactic, and thus the proof script, fails if it doesn't change the proof state. For our proof script, it is perfectly acceptable that tactics do not always change the proof state.

Next, *clarsimp* is used. This tactic applies *simp* and *clarify*. The *simp* tactic unfolds the definitions used and applies other simplifications. *clarify* performs all the safe steps that do not split the goals into subgoals.

If the goal is still not proved after *clarsimp*, the *safe* tactic is applied. This tactic applies all safe rules, including those that split the goal. From this point on it is possible that there is more than one subgoal in the proof state.

If there are still subgoals left after *safe*, then we have to resort to tactics that perform both safe and unsafe steps. The *auto* tactic is used for this. In some situations, the *auto* tactic cannot prove the goal if there is an existential quantifier on the top-level of the conclusion, but is able to prove the goal if the existential quantifier is first replaced by a schematic variable. To prove even

those goals, the choice tactical is used (`auto | rule exI`). This states that in the event that *auto* fails, the elimination of the existential quantifier from the conclusion (`rule exI`) should be tried, before giving *auto* another chance. If both fail, then `auto | rule exI` fails.

The `+` tactical indicates that the tactic should be repeated. It is used in combination with the `?` tactical to apply the tactic zero or more times, as long as there is progress.

The *by* command does not only apply tactics, but also applies the *assumption* tactic afterwards and uses backtracking to try to prove the goal.

Next, two examples are discussed. The first example shows a situation where *auto* is not able to prove the goal without the extra `rule exI`. The second example demonstrates the importance of backtracking and demonstrates that the order in which `auto` and `rule exI` are applied does matter in some situations.

auto* and *rule exI Consider the following goal:

$$\bigwedge a\ b.\ [\![a \sim= b;\ f\ b]\!] \implies \text{EX } c.\ (\,(c = a \longrightarrow f\ a) \wedge (c \sim= a \longrightarrow f\ c)\,)$$

Until now, we have only dealt with goals without any bound variables on the top-level of the goal, but variables a and b of this goal are bound by a universal quantification at the top-level. The assumptions and the conclusion are both within the scope of this universal quantifier.

The goal is valid, this can be proved by instantiating the existential quantifier with b. The *force* tactic can prove this goal, but the *auto* tactic cannot. If the existential quantifier is replaced by a schematic variable (`rule exI`), then we arrive at:

$$\bigwedge a\ b.\ [\![a \sim= b;\ f\ b]\!] \implies ((?c\ a\ b) = a \longrightarrow f\ a) \wedge ((?c\ a\ b) \sim= a \longrightarrow f\ (?c\ a\ b))$$

This goal is still valid, but now *auto* can prove this. Because the *exI* rule is in the set of rules of default introduction rules, we suspect that *auto* fails to prove the original goal because of the order in which rules are applied.

In most of our examples it makes no difference whether variables are free, or bound at the outer level of the goal. But for this goal, it does make a difference for the *auto* tactic. If a and b are free variables instead of bound variables, then the *auto* tactic is able to prove the goal without having to apply the *exI* rule first. Unfortunately we cannot provide an explanation for this behaviour.

backtracking Consider the following goal:

$$[\![f\ (3{::}nat);\ f\ 4]\!] \implies \text{EX } x.\ \ f\ x\ \ \&\ \ f\ (x-1)$$

f is a function from *nat* to *bool*. Because we have explicitly stated the type of term 3, Isabelle is able to infer the type of f and 4. The *auto* tactic can prove this goal, *auto* is apparently able to somehow determine a correct instantiation for the existential quantification, hence the goal can be proved using the proof script. If instead of `auto`, `rule exI` is applied to the original goal, 3 is substituted for $?x$, and we arrive at:

$$[\![f\ (3 :: nat);\ f\ 4]\!] \implies f\ ?x\ \ \&\ \ f\ (?x-1) \tag{4.1}$$

The existential quantifier has been eliminated, and the bound variable has become a schematic variable. This goal is still valid, and can be proved by substituting the term 4 for the schematic variable $?x$. If we supply term 4 to be substituted for the schematic variable in `exI`, then the simplifier can do the rest: `apply(rule_tac x = 4 in exI, simp)`. But, obviously, this is not a viable option for the automated proof script, because the proof script has to prove arbitrary goals. If *auto* is applied to the goal with the schematic variables (4.1), we arrive at:

$$[\![f\ (3 :: nat);\ f\ 4]\!] \implies f\ 2$$

This obviously unprovable. To illustrate what goes wrong we will show, in small steps, what auto does behind the scenes.

First the conjunction in the conclusion is eliminated: `apply(rule conjI)`, the goal is split into the following two new subgoals:

$$[\![f\ (3 :: nat)\,;\ f\ 4]\!] \implies f\ ?x \tag{4.2}$$
$$[\![f\ (3 :: nat)\,;\ f\ 4]\!] \implies f\ (?x - 1) \tag{4.3}$$

The first subgoal is proved using the *assumption* tactic (`apply(assumption)`); 3 is substituted for $?x$, which changes the second subgoal to:

$$[\![f\ (3 :: nat)\,;\ f\ 4]\!] \implies f\ (3 - 1)$$

And this is where it goes wrong. Unification finds more than one candidate for the substitution for the schematic variable, the tactic has to choose one, and picks the first by default. In this case, an unfortunate choice was made, one that proved the first subgoal but rendered the second subgoal unprovable. Even though the *assumption* tactic is applied to only the first goal, this example illustrates that it can still effect other subgoals. Luckily, backtracking is available in Isabelle. If the command `back` is used at this point, Isabelle tries the next candidate for $?x$ in (4.2), which is 4. Using this substitution, both goals can be proved.

The *by* command uses backtracking automatically if the goal cannot be proved. So, if `by(auto)` is used instead of `apply(auto)`, then goal (4.1) can be proved automatically.

Proof script B

```
lemma loc_ass_3a_stat_2: "
  ALL (s :: state).
    ALL (b :: component).
      lab_2 b s  ⟶
        wlp_stat_2 b (ass_3a b) s"
```

```
apply(clarify)?
apply(insert lab_2_inv_0a, rotate_tac -1)?
apply(erule_tac x = "s" in allE, rotate_tac -1)?
apply(erule_tac x = "b" in allE, rotate_tac -1)?
apply(insert lab_2_ass_2a, rotate_tac -1)?
apply(erule_tac x = "s" in allE, rotate_tac -1)?
apply(erule_tac x = "b" in allE, rotate_tac -1)?
apply(insert lab_2_ass_2b, rotate_tac -1)?
apply(erule_tac x = "s" in allE, rotate_tac -1)?
apply(erule_tac x = "b" in allE, rotate_tac -1)?
by( (clarify | simp | rule conjI | force)+ )?
```

The script starts the same way as *Proof script A*: the *clarify* tactic performs the obvious reasoning steps. Implication and universal quantification are removed from the conclusion and the relevant axioms are introduced and instantiated.

The relevant difference between the two proof scripts is that *Proof script B* applies the introduction rule *conjI* earlier in the proof than *Proof script A* does. The *conjI* rule is a safe rule that splits the goal into subgoals, one for each conjunct on the top-level of the conclusion. By applying the *conjI* rule before other safe rules that split the goal, we achieve that *clarify* and *simp* can work with less complex conclusions, which works better for many goals with conjunctive conclusions.

Furthermore, the *force* tactic is used instead of *auto* because we'd like to treat the goals separately: in contrast to *auto*, the *force* tactic is only applied to a single subgoal. The tactics *rule conjI*, *clarify*, and *simp* are also applied only to the first subgoal. Each time a subgoal is proved, the next subgoal moves up in the list.

This last combination of tactics has the following effect on the proof state:

1. Apply *clarify* and *simp* to the first subgoal. Then go to 2.
2. Apply rule *conjI* to the first subgoal. If this succeeds (the proof state has changed) then go to 1. If not, then go to 3.
3. Apply *force* to the first subgoal. If this succeeds then the subgoal is proved and removed. If there are no more subgoals left, then the original proof obligation is proved, if not, then it goes back to 1 (another subgoal moved to the head of the list). If the subgoal cannot be proved by force, then the proof obligation cannot be proved automatically with this proof script.

Alternatives

Automated tactics There are various tactics available in Isabelle that do automated reasoning. The tactics that do the main work in the proof scripts are *auto* and *force*. We have experimented with alternatives like *blast*, *best*, *fast*, and even *arith*, but it turned out that none of them were as effective as *auto* and *force*, at least not for our current set of examples.

Elimination of implication and universal quantification in the conclusion The first step in the proof scripts is the elimination of the implications and universal quantifiers in the conclusion. Initially the tactic `apply(intro impI allI)` was used for this. This tactic applies the rules *impI* (introduction rule for implication) and *allI* (introduction rule for universal quantification) repeatedly, as long as there is progress.

There are proof obligations where *false* is in the antecedent of an implication and thus in the assumptions of the goal after the elimination of implication and universal quantification. Because *false* is stronger than any arbitrary term, there is no need to introduce and instantiate axioms if *false* is in the premises. If `apply(clarify)` is used instead of `apply(intro impI allI)`, then a goal with *false* in the antecedent is proved by this first step, because *clarify* also applies other safe rules and one of them is able to prove goals with *false* in the assumptions/antecedent. In such case, the axioms are never employed, and the definitions of assumptions, statements and guards are never unfolded. This approach speeds up the proof considerably if there are a large number of proof obligations with *false* in an antecedent of an implication; this is one of the reasons why we adjusted the scripts.

Order *clarify* and *simp* The tactics *clarsimp*, *auto*, and *force* internally use the tactics *simp* and *clarify*. It turned out that the order in which *simp* and *clarify* are applied can make a significant difference.

The Isabelle/HOL tutorial [NPW02] describes the *clarsimp* tactic as: "a method that interleaves *clarify* and *simp*". Reading this description, one might expect that the *clarify* tactic is applied before the *simp* tactic, but it is actually the other way around, *simp* is applied first. Experiments have taught us that the performance of *Proof script A* is better if *simp* is applied first and that the performance of *Proof script B* is better if *clarify* is applied first. Originally, *clarsimp* was used for both proof scripts, but we have changed it to *clarify* | *simp* in *Proof script B*.

Both the time it takes to prove the proof obligations, and the number of proof obligations that can be proved automatically, is influenced by this order. We have run into goals that can only be proved automatically if *clarify* is applied first, before unfolding the definitions. By applying *clarify* as the very first step in the proof, we have solved this problem for our current set of examples.

Next, we discuss an example of a goal that can be proved automatically if *clarify* is applied first, but cannot be proved automatically if *simp* is applied first. Consider the following goal:

$$[\![\sim b \,;\, \sim (\,(\mathit{if}\ a\ \ \mathit{then}\ \ \sim b\ \ \mathit{else}\ \ \sim c) \mid c\,)]\!] \implies \mathit{false}$$

If *simp* is applied to this goal then we arrive at goal (4.4), and if *clarify* is applied to this goal then we arrive at goal (4.5).

$$[\![\sim b \,;\, \sim (\mathit{if}\ a\ \mathit{then}\ \sim b\ \mathit{else}\ \sim c)\ \&\ \sim c\ ]\!] \implies \mathit{false} \tag{4.4}$$

$$[\![\sim b \,;\, \sim c]\!] \implies \mathit{if}\ a\ \mathit{then}\ \sim b\ \mathit{else}\ \sim c \tag{4.5}$$

Both goals are still valid, but the first goal cannot be proved by *auto*, nor by *force*. The second goal can be proved by both *auto* and *force*.

The original goal cannot be proved by *auto*, but can be proved by *force*. The example above shows why: the *auto* tactic starts by applying *simp*, and the *force* tactic applies *clarify* first.

4.5 Adding Isabelle support to the tool

Initially, the tool only generated proof obligations and their automatic proof scripts for PVS. Support for Isabelle has been added to the tool. In Section 4.3 we have discussed how programs, annotation and proof obligations can be modelled in Isabelle. In Section 4.4 we have discussed the default proof strategy for the Isabelle proofs. In this section we give an overview of the changes made to the tool in order to add support for Isabelle.

The tool reads a file that defines the structure of the program, this file is parsed and an internal representation of the program and the proof obligations is created. The tool then uses this internal representation to generate the proof obligations and the proof scripts for a theorem prover.

The internal representation uses separate classes for the representation of each type of proof obligation. The following fragment of Python code of the tool shows the class that is used to represent the proof obligation for the local correctness of $\{P\}S\{Q\}$, where S is an atomic statement.

```
class SimpleLocalCorrectnessProof(LocalCorrectnessProof):

  def to_pvs(self):
      P = self.P
      S = self.statement
      Q = self.assertion

      lines = self.pvs_proof_start()
      lemma_dependencies = S.dependent_declarations
      lines.append(pvs.forall(lemma_dependencies))
      lines.append('%s =>' % P.to_pvs())
      lines.append('%s(%s)(s)' % (S.to_pvs(), Q.to_pvs_no_state()))
      return indent_lines(lines)

  def to_isabelle(self):
      P = self.P
      S = self.statement
      Q = self.assertion

      lines = self.isabelle_proof_start()
      lemma_dependencies = S.dependent_declarations
      lines.append(isabelle.forall(lemma_dependencies))
      lines.append('%s -->' %P.to_isabelle())
      lines.append('%s (%s) s"' % (S.to_isabelle(), Q.to_isabelle_no_state()))
      return indent_lines(lines)
```

The function `to_isabelle()` has been added to translate this particular type of proof obligation to an Isabelle representation. A similar function is added to each of the classes that represents a type of proof obligation. The representation of axioms, definitions and the proof scripts are handled in a similar way.

A notable difference between Isabelle and PVS is the way proof obligations are related to their proofs. In Isabelle, proof obligations and their proofs are combined: in the Isabelle theory, a proof obligation is directly followed by the proof. An Isabelle theory is read from top to bottom, and definitions and theorems can only be used if they have already been declared and proved previously. In PVS, the proof obligations are separated from their proofs. Every theory consists of a file containing the definitions and the proof obligations, and another file containing the proofs of the proof obligations.

Only minor changes to the tool were required to combine the proof obligations and proofs for the Isabelle.

4.6 Experiments

We have experimented with various, mostly small, annotated algorithms. Correctness of some algorithms can be verified completely automatically using the tool and Isabelle, others need interaction with the user – the user has to prove some of the proof obligations manually. In this section a summary of the outcome of the experiments is given.

The following algorithms have been modelled and verified with Isabelle:

- a coarse-grained and a finer-grained solution of a simple election protocol [FvG99] (see also Section 3.2);
- several variants of "The Initialization Protocol" for two and three components [FvG99] (see also Section 3.3);
- monitored phase synchronization [FvG99];
- mutual exclusion using a ticket algorithm [PN02];
- mutual exclusion using Peterson's algorithm for two processes [FvG99];
- mutual exclusion using semaphores [PN02];
- parallel linear search [FvG99];
- wait-free consensus protocol [Moo02];
- wait-free handshake register [Hes98].

The results are shown in Table 4.1 [1]. For each of the algorithms the number of proof obligations generated, the number of proof obligations that can be proved automatically and an indication of the time it takes to replay the complete proof using *Proof script A* or *Proof script B* is given. The proofs that could not be proved automatically are proved manually, and the failed attempts to prove them automatically are not included in the runtime.

The cases where manual proofs are required are displayed in bold. For all these cases, with the exception of the second variant of "The Initialization Protocol" for three components, the set of proof obligations that require manually proofs is the same for both proof scripts. For the second variant of "The Initialization Protocol" for three components, there are six proof obligations that can be proved with *Proof script A*, but cannot be proved using *Proof script B*.

Even though more can be proved using *Proof script A*, *Proof script B* is faster for some algorithms. This difference is most noticeable for the proof of the "Wait-free handshake register" algorithm.

4.7 Evaluation of the Isabelle proofs

In this section the Isabelle proofs of the various algorithms discussed in the previous section are evaluated. The problematic lemmas are discussed; for each problematic lemma we attempt to determine why it is problematic and what can possibly be done about it.

[1] An Intel Xeon 2.8GHz processor is used. Only a small fragment of the 4GB memory available is actually used.

Algorithm	*# Obligations*	*# Auto*		*Time (sec)*	
		A	B	A	B
A simple election protocol (coarse-grained)	27	27	27	5	2
A simple election protocol (finer-grained)	60	60	60	5	5
Init. protocol for two components (Original)	216	216	216	18	16
Init. protocol for two components (Variant 1)	676	676	676	124	114
Init. protocol for two components (Variant 2)	75	75	75	12	12
Init. protocol for two components (Variant 3)	75	75	75	9	6
Init. protocol for three components (Original)	1278	1278	1278	321	288
Init. protocol for three components (Variant 1)	2619	2619	2619	2143	1947
Init. protocol for three components (Variant 2)	236	236	**230**	154	146
Init. protocol for three components (variant 3)	236	236	236	118	100
Monitored phase synchronization	93	**69**	**69**	11	10
Mutual exclusion using a ticket algorithm	71	**59**	**59**	12	11
Mutual exclusion using Peterson's algorithm	51	51	51	4	4
Mutual exclusion using semaphores	13	13	13	1	1
Wait-free consensus protocol	26	26	26	1	1
Wait-free handshake register	178	**172**	**172**	1664	572
Parallel linear search	17	17	17	1	1

Table 4.1: Results of the experiments

4.7.1 Mutual exclusion using a ticket algorithm

In this section we address the problems encountered when verifying "Mutual exclusion using a ticket algorithm". We discuss several (generalised) problems we encountered when trying to prove the proof obligations.

Consider the following goal:

$$[\![\, \text{ALL } c.\ \mathit{turn}\ s\ c < \mathit{num}\ s\,;\ \mathit{turn}\ s\ \mathit{c1} = \mathit{num}\ s \,]\!] \quad \Longrightarrow \quad \mathit{num}\ s = 0$$

where *turn* is of type: *component* ⇒ *state* ⇒ *nat*. This goal is obviously valid as the conjunction of the assumptions is equivalent to *false*. We can prove this by instantiating the universal quantifier with term *c*1 , and applying simplification afterwards: `by(erule_tac x = "c1" in allE, simp)`. Unfortunately the automated tactics (tactics that do automated reasoning, using the default settings and rules), and thus the proof scripts, cannot prove this goal. Each tactic that does automated reasoning fails on this goal. The problem is that the quantifier is not instantiated or simplified. There are various ways to use the universal quantifier in the assumptions. We first discuss various options, and a more detailed explanation follows. The options are:

1. Instantiate the quantifier.
2. Weaken ALL *x*. *P x* to *P ?x* . The unsafe (destruction) rule *spec* can do this strengthening of the goal. This is possible because types are non-empty.
3. Move the quantifier from the assumptions to the conclusion. To this end we define the (elimination) rule:

   $$[\![\, \text{ALL } x.\ ?P\ x\,;\ (\sim ?Q \implies \text{EX } x.\ \sim ?P\ x) \,]\!] \quad \Longrightarrow \quad ?Q$$

   This rule states that ALL *x*. *?P x* in the assumptions can be moved to the conclusion as EX *x*. ∼ *?P x* , if the negation of the original conclusion is added as an assumption. Even though this transformation is safe, the existential quantification in the conclusion may be treated different from the universal quantification in the assumptions.

Re 1 There are no tactics available to automatically instantiate the quantifier with terms, using some heuristics or simply using brute force to find candidates for the instantiation. When we're proving the goal interactively, the instantiation with $c1$ is obvious, but this is very specific for this goal; we cannot use this in a proof script.

Re 2 After applying the elimination rule we arrive at:

$$[\![\, turn\ s\ ?c\ <\ num\ s\,;\ turn\ s\ c1\ =\ num\ s\,]\!] \quad \Longrightarrow \quad num\ s\ =\ 0 \tag{4.6}$$

The automated tactics cannot prove this goal. We expected this to be a typical goal for the *arith* tactic, but this tactic is also not able to prove this goal.

Next we experiment with different tactics to try to prove this goal. First we apply the tactic `apply(subgoal_tac "~turn s c1 < num s")`. This transforms (4.6) into the following two subgoals:

$$[\![\, turn\ s\ ?c\ <\ num\ s\,;\ turn\ s\ c1\ =\ num\ s\,;\ \sim\ turn\ s\ c1\ <\ num\ s\,]\!] \quad \Longrightarrow \quad num\ s\ =\ 0 \tag{4.7}$$

$$[\![\, turn\ s\ ?c\ <\ num\ s\,;\ turn\ s\ c1\ =\ num\ s\,]\!] \quad \Longrightarrow \quad \sim\ turn\ s\ c1\ <\ num\ s \tag{4.8}$$

If *auto* is used at this point, the second subgoal (4.8) is proved, but the first subgoal (4.7) is transformed back to its original (4.6) — removing the additional assumption. The simplifier is to blame for this; each tactic that uses the simplifier has this as result. Part of the simplifier trace is:

```
SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
[| ALL c. turn s c < num s; turn s c1 = num s;
   ~ turn s c1 < num s |] ==> num s = 0
Applying instance of rewrite rule:
turn s c1 == num s
Rewriting:
turn s c1 == num s
Applying instance of rewrite rule "HOL.order_less_irrefl":
?x1 < ?x1 == False
Rewriting:
num s < num s == False
Applying instance of rewrite rule "HOL.simp_thms_8":
~ False == True
Rewriting:
~ False == True
Adding rewrite rule:
turn s ?x1 < num s == True
```

Blast does not use the simplifier and is able to prove the first subgoal (4.7). The *?c* variable in the second subgoal (4.7) is then bound to *c1* , and the simplifier can prove this last subgoal.

Unfortunately, this cannot be used in the default proof script, as it is too specific for this goal (the *subgoal_tac*, that is).

Re 3 After the quantifier has been moved from the assumptions to the conclusion, we arrive at:

$$[\![\, turn\ s\ c1\ =\ num\ s\,;\ num\ s\ \sim\!=\ 0\,]\!] \quad \Longrightarrow \quad \text{EX } c.\ \sim\ turn\ s\ c\ <\ num\ s$$

This goal cannot be proved using the tactics that do automated reasoning, using the default settings. After applying the obvious steps (`apply(clarify, rule exI)`) we arrive at:

$$[\![\, turn\ s\ c1\ =\ num\ s\,;\ 0\ <\ num\ s\,]\!] \quad \Longrightarrow \quad \sim turn\ s\ ?c\ <\ num\ s$$

When *clarify* (or any tactic that uses *clarify*) is applied to this goal, then $turn\ s\ ?c\ <\ num\ s$ is moved to the assumptions, leaving *false* in the conclusion. This new goal can still not be proved automatically.

We use the fact that $\sim a < b$ is equivalent to $(a \sim= b) \longrightarrow b < a$ and define and prove the introduction rule:

$$(?P\ \sim=\ ?Q\ \Longrightarrow\ ?Q\ <\ ?P) \quad \Longrightarrow \quad \sim\ ?P\ <\ ?Q \qquad \text{(LT\_CONC)}$$

After applying this rule (`apply(rule LT_CONC)`) we arrive at the goal:

$$[\![\, turn\ s\ c1\ =\ num\ s\ ;\ 0\ <\ num\ s\ ;\ turn\ s\ ?c\ \sim=\ num\ s \,]\!] \quad \Longrightarrow \quad num\ s\ <\ turn\ s\ ?c$$

This goal can be proved by substituting $c1$ for $?c$, and this is exactly what the automated tactics do. The reason the instantiation for this goal is found automatically, in contrast to the original, is that unification succeeds here.

Unfortunately, this solution is too specific for this goal. Note that even if the rule *LT_CONC* would be added to set of default introduction, then in order to prove this goal we would still have to ensure that it is applied before the rule that moves the negated formula in the conclusion to the assumptions.

In conclusion, we have no other choice but to prove the proof obligations of this form manually.

Consider the following goal:

$$(f :: nat \Rightarrow nat)\ a\ =\ g\ b \quad \Longrightarrow \quad g\ b\ =\ f\ ?a$$

For the sake of readability, type information is left out from this point on. This goal is valid with $?a\ :=\ a$, but this cannot be proved using the automated tactics. It can be proved automatically after applying symmetry of $=$: `apply(rule sym)`

$$f\ a = g\ b \quad \Longrightarrow \quad f\ ?a = g\ b$$

The *assumption* tactic proves this resulting goal: the tactic unifies the conclusion with the assumption and finds a correct instantiation that proves the goal. The *sym* rule cannot be a default introduction rule because it would result in situations where the left and right hand side of an equation are swapped indefinitely.

Next we try some variations of the goal. First the left and right hand side of both equations are swapped:

$$g\ b = f\ a \quad \Longrightarrow \quad f\ ?a = g\ b$$

The *auto* tactic can prove this goal, which was a bit of a surprise at first. It is actually the simplifier that proves the goal, the output of the simplifier trace is:

```
 SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
(g::nat => nat) (b::nat) = (f::nat => nat) a ==> f (?a::nat) = g b
Adding rewrite rule:
(g::nat => nat) (b::nat) == (f::nat => nat) a
Applying instance of rewrite rule:
(g::nat => nat) (b::nat) == (f::nat => nat) a
Rewriting:
(g::nat => nat) (b::nat) == (f::nat => nat) a
```

The simplifier creates a rewrite rule that states that $g\ b$ can be rewritten into $f\ a$, which is then applied and we arrive at a goal with $f\ ?a\ =\ f\ a$ in the conclusion, which is trivially true. We did not get a similar result for the original goal because the assumption of the original goal was $f\ a\ =\ g\ b$, and a rewrite rule that rewrites $f\ a$ into $g\ b$ was added, but there was no $f\ a$ in the conclusion.

Next, consider the following goal:

$$f\ a\ =\ q\quad \Longrightarrow\quad q\ =\ f\ ?a \tag{4.9}$$

The only difference between this goal and the original goal is that the term $g\ b$ is replaced by the variable q; both terms are of type *nat*. This resulting goal can be proved automatically using the *simplifier*. The reason that this goal can be proved, in contrast to the original, is that for this goal a rewrite rule is added that rewrites q into $f\ a$. Apparently a distinction is made between terms that are just variables, and terms that are not variables.

4.7.2 Initialization protocol

We have encountered three types of problems when verifying the variants of the initialization protocol for two and three components:

1. Isabelle's automated tactics do not use the fact that X, Y and Z are the only possible values a term of type *component* can have, for the solutions for three components.
2. Isabelle's automated tactics do not always simplify (and use) *if-then-else* constructs.
3. There are proof obligations that can be proved using *Proof script A*, but cannot be proved with *Proof script B*.

Next, these three problems are discussed.

component enumeration type The *component* type for the solutions for three components is defined as `datatype component = X | Y | Z`. The only three possible values for a term of type *component* are X, Y and Z. Even though Isabelle automatically defines a theorem stating just this, the automated tactics of Isabelle do not use this fact. As a consequence, several proof obligations cannot be proved automatically without adding this fact to the assumptions of the proofs. Those proof obligations are of the form:

$$[\![\, (i :: component) \sim= X\ ;\ i \sim= Y\ ;\ r\ i \,]\!]\quad \Longrightarrow\quad r\ Z$$

The proof of this goal needs the fact that, because i is not equal to X, and is not equal to Y, Z is the only possible value for i. The following theorem is added as a system invariant of the program:

$$\text{ALL } (c :: component).\ \ c = X\ \mid\ c = Y\ \mid\ c = Z$$

Using this extra invariant, the proof obligations can be proved automatically using the proof scripts.

If-then-else construct The second variant of the solution for two components uses *if-then-else* constructs to select the correct assertion, assignment or guard for the specific component. Several proof obligations using this *if-then-else* construct could not be proved automatically.

Consider the following goal:

$$[\![\, if\ b\ then\ c\ else\ d\ ;\ \sim c \,]\!]\quad \Longrightarrow\quad d$$

As c is *false*, b can only be *false*, hence d is *true*. This goal cannot be proved using the automated tactics. To prove this goal we'd like to rewrite the *if-then-else* construct on booleans. The following equalities can be used:

$$\textit{if } b \textit{ then } c \textit{ else } d \quad \equiv \quad (b \longrightarrow c) \wedge (\sim b \longrightarrow d) \tag{4.10}$$

$$\textit{if } b \textit{ then } c \textit{ else } d \quad \equiv \quad (b \wedge c) \vee (\sim b \wedge d) \tag{4.11}$$

While equation (4.10) is most efficient for *if-then-else* constructs in the conclusion, equation (4.11) is most efficient for *if-then-else* constructs in the assumptions. An introduction rule that uses equation (4.10) to replace *if-then-else* constructs *in the conclusion* is in the set of default introduction rules.

if-then-else constructs *in the assumptions* are not simplified in this way. This is most likely because it could lead to a large number of disjunctions and implications in the assumptions. Disjunctions and implications in the assumptions can lead to a large number of subgoals as the *safe* tactic tends to split every goal into two subgoals, for each implication and disjunction in the assumptions. (4.11) is more efficient for *if-then-else* constructs in the assumptions because we only get one disjunction in the assumptions, where (4.10) would add two implications: the number of possible splits is linear in the number of *if-then-else* constructs in the assumptions, instead of exponential.

We add the following destruction rule to the set of default rules:

$$(\textit{if } a \textit{ then } b \textit{ else } c) \quad \Longrightarrow \quad (a \;\; \& \;\; b) \;\; | \;\; (\sim a \;\; \& \;\; c) \qquad \text{(split\_disj)}$$

With this extra rule, all the problematic proof obligations can be proved automatically. Note that even though this rule is used as a destruction rule (it transforms one of the assumptions), it is actually safe, as it transforms the assumption into a logically equivalent assumption.

We were surprised that even the following goal could not be proved automatically *without* this rule:

$$[\![\sim p \,;\, \sim q \,;\, \textit{if } b \textit{ then } p \textit{ else } q]\!] \quad \Longrightarrow \quad \textit{false}$$

The simplifier should be able to prove this: both p and q in the *if-then-else* construct can be simplified to *false*, and *if b then false else false* can be rewritten to *false*, using the rewrite rule *split_cancel*, which is in the set of default simplifier rules. Even though the simplifier adds rewrite rules from p and q to *false*, they are not automatically applied to the assumption. In [NPW02, Section 3.1.9] it is indeed mentioned that the simplifier only simplifies the *if* condition, and does not simplify the *then* part and the *else* part.

Proof script A versus Proof script B After the changes mentioned above, all proof obligations can be proved automatically using *Proof script A*, but there are a few proof obligations of the second variant of the solution for three components that cannot be proved with *Proof script B*. Next these proof obligations are discussed.

After simplification the goals are of the form:

$$[\![\text{ALL } c.\; c = X \;|\; c = Y \;|\; c = Z \,;\, i \sim= i0 \,;\, i \sim= Y \,;\, i \sim= X \,;\, i0 \sim= Y]\!] \quad \Longrightarrow \quad i0 = X$$

This goal is valid because i can only be Z, hence $i0$ can only be X. *auto* and *blast*, and thus *Proof script A*, can prove this goal. But, *force*, and thus *Proof script B*, cannot. Only after the universal quantification is copied and instantiated with i, or case analysis is applied on both i and $i0$, *force* is able to prove this goal. Both solutions are too specific for this goal and cannot be used in a generic proof script.

In conclusion, if we use *Proof script B* as default proof script for this algorithm, then we have to prove the proof obligations of this form manually.

4.7.3 Monitored phase synchronization

24 of the proof obligations generated for this algorithm cannot be proved automatically using Isabelle, using either *Proof script A* or *Proof script B*. Next we discuss the problems encountered.

Consider the following goal:

$$[\![\text{ALL } d.\ (x\ d \leq m) = (d = f)\,;\ c0 = f\,;\ \text{ALL } d.\ m \leq x\ d]\!] \implies x\ c0 \leq x\ d0$$

If the first universal quantifier is instantiated with $c0$, and the second quantifier is instantiated with $d0$, then we get $(x\ c0) \leq m \leq (x\ d0)$ in the assumptions. The conclusion is then trivially *true* because of the transitivity of $\leq$. This can be proved using the *simp* or *arith* tactic. Even though the number of candidates for the instantiation the quantifiers is very limited, none of the automated tactics can prove the original goal. Since the proof requires only a single instantiation of both quantifiers, we can try to weaken the quantifiers to schematic variables using `apply(drule spec)+`. The resulting goal is:

$$[\![(x\ ?d \leq m) = (?d = f)\,;\ c0 = f\,;\ m \leq x\ ?d2]\!] \implies x\ c0 \leq x\ d0$$

This goal is still valid with $?d := c0$ and $?d2 := d0$. Isabelle uses only unification to find candidates for the substitution of schematic variables, but unification is not able to find any candidates for $?d$ and $?d2$, and the goal can still not be proved automatically. We have no other choice but to prove the proof obligations of this form manually.

Next we give another example of a goal that could not be proved using the proof scripts. Consider the following goal:

$$\text{ALL } d.\ x < f\ d \implies Suc\ x \leq f\ d0$$

Where *Suc* is the successor function for naturals. This goal can be proved by instantiating the quantifier with term $d0$ and applying *simp* afterwards, but cannot be proved automatically.

$Suc\ x \leq f\ d0$ is equivalent to $x < f\ d0$. The lemma *Suc_le_eq*, which is available (and proved) in the HOL library, states this. The simplifier can prove the goal when this lemma is added to the set of default rewrite rules, we give the relevant part of the simplifier trace:

```
Adding rewrite rule "Nat.Suc_le_eq":
Suc ?m1 <= ?n1 == ?m1 < ?n1
SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
ALL d. x < f d ==> Suc x <= f d0
Adding rewrite rule:
x < f ?d1 == True
Applying instance of rewrite rule "Nat.Suc_le_eq":
Suc ?m <= ?n1 == ?m1 < ?n1
Rewriting:
Suc x <= f d0 == x < f d0
Applying instance of rewrite rule:
x < f ?x1 == True
Rewriting:
x < f d0 == True
```

Not only is the simplifier able to rewrite the conclusion, it is also able to finish the proof using rewrite rules derived from the assumptions. Because both *auto* and *force* use the simplifier, both proof scripts are able to prove this goal if the lemma *Suc_le_eq* is added to the set of default rewrite rules.

If we apply the simplifier, but disallow it from using rewrite rules derived from the assumptions, then we get:

$$\text{ALL } d.\ x < f\ d \implies x < f\ d0$$

If this quantifier is weakened to a schematic variable we get the following goal:

$$x < f\ ?d \quad \Longrightarrow \quad x < f\ d0$$

This goal can be proved using the *assumption* tactic. This time unification finds the required substitution for $?d$: $?d := d0$.

Unfortunately the discussed solutions for the problematic proof obligations for this algorithm are too specific for a general purpose proof script, and we have no choice but to prove the 24 problematic proof obligations manually.

4.7.4 Wait-free handshake register

Six proof obligations of the wait-free handshake register algorithm cannot be proved automatically using *Proof script A* and *Proof script B* because relatively complex instantiations of universal quantifiers are required for their proofs, which are not found by unification. We have no other choice but to prove the six lemmas manually.

It takes a relatively long time to verify the proof obligations using *Proof script A* because there are a large number of implications and disjunctions in the assumptions. This was the initial reason for creating *Proof script B*. First, we explain why it takes *Proof script A* such a long time to prove proof obligations with a large number of implications and disjunctions in the assumptions. Afterwards, we discuss why *Proof script B* is much faster for this algorithm, and we give an example that illustrates this difference.

Both *auto* and *force* start by applying *simp* and *clarify* and then, if the goal is still not proved, *safe* is applied. The *safe* tactic removes implications and disjunctions from the assumptions by splitting the goal into two subgoals, for each implication and disjunction in the assumptions, as long as the goal cannot be proved using other safe rules.

So if there are n assumptions with implications or disjunctions after simplification, then the goal is split into 2^n subgoals, unless other safe rules can prove the goal before splitting.

The conclusion of many of the proof obligations of this algorithm is conjunctive and the *safe* tactic eventually splits the goals into subgoals, one for each conjunction in the conclusion. *Proof script B* splits the conjunctions earlier in the proof than *Proof script A* does and applies *clarify* and *simp* repeatedly to every subgoal as long as they have an effect on the subgoals. The effect is that in many cases, the implications and disjunctions in the assumptions do not have to be removed, so the goal does not have to be split into 2^n subgoals, which makes the proof much faster. The goal does not have to be split into many subgoals in these cases because *simp* and *clarify* are applied to simpler conclusions, the conjuncts of the original conclusion, before other rules that split the goal are applied. This way they can be proved, before the rules that split the rules are applied.

Next, we use an example to illustrate this. Consider the following goal:

$$\begin{array}{l} [\![a \longrightarrow b; c \longrightarrow d; e \longrightarrow i; k \longrightarrow l; m \longrightarrow p; q \longrightarrow r; s \longrightarrow t; v \longrightarrow w; \\ \quad x \longrightarrow y \longrightarrow z; u \leq f\ (g\ n)]\!] \quad \Longrightarrow \\ \quad\quad (g\ n = h\ n \longrightarrow u \leq f\ (h\ n))\ \&\ (g\ n = j\ n \longrightarrow u \leq f\ (j\ n)) \end{array}$$

If we apply *Proof script A* to this subgoal, we get the following scenario:

- *clarsimp* has no effect;
- *safe* needs 350 seconds and splits the goal into 768 subgoals;
- *auto* proves all 768 subgoals in about 10 seconds. It is actually the simplifier that proves them.

And *Proof script B* has the following effect:

- *clarify* and *simp* have no effect;
- *rule conjI* splits the goal into two subgoals, one for each of the conjuncts in the conclusion;
- for each of the two subgoals:
  - *clarify* eliminates the implication in the conclusion (ex. $g\ n = h\ n$ is moved to the assumptions) ;
  - *simp* proves the subgoal. *simp* uses the equality that was just added to the assumptions as rewrite rule, i.e. $g\ n \mapsto h\ n$.

Proof script B needs less than a second for the complete proof.

If we use the symmetry of $=$ and swap the left hand side and the right hand side of the equation in the conclusion, then *simp* can actually prove this goal without ever splitting the goal into subgoals. If *simp* is applied to the goal

$$\begin{array}{l}[\![a \longrightarrow b; c \longrightarrow d; e \longrightarrow i; k \longrightarrow l; m \longrightarrow p; q \longrightarrow r; s \longrightarrow t; v \longrightarrow w; \\ \quad x \longrightarrow y \longrightarrow z; u \leq f\ (g\ n)]\!] \implies \\ \qquad (h\ n = g\ n \longrightarrow u \leq f\ (h\ n))\ \&\ (j\ n = g\ n \longrightarrow u \leq f\ (j\ n))\end{array}$$

then a congruence rule of implication is used, and $h\ n$ and $j\ n$ in the conclusion are rewritten to $g\ n$, and *simp* simplifies the conclusion to *true*. This is another example that illustrates that symmetry can be problematic.

There is often a trade-off between effort and time, in this case between the number of proof obligations that can be proved automatically and the runtime of the proof script. There are situations where there user has to choose between waiting for the proof script to terminate (possibly successful) and the time and effort required to prove the proof obligation manually. In this section we have focused on the difference in the amount of time that both proof scripts need to prove the proof obligations of this algorithm. The proof scripts can prove the same proof obligations for this algorithm, but *Proof script B* can prove many of them much faster.

The reader may get the impression that the difference is marginal, that the user should just be a little more patient when using *Proof script A*. But, it is important to realise that both the time and the number of subgoals increases exponentially with the number of implications and disjunctions in the assumptions. Adding one extra implication to the assumptions of the goal above, increases the runtime of *Proof script A* with about 2000 seconds, while *Proof script B* can still prove the goal in less than a second.

4.8 Conclusions

In this chapter we have given a short introduction to Isabelle and (automated) verification with Isabelle. We have discussed how programs, annotation and proof obligations can be modelled in Isabelle, and we have discussed a generic automated proof strategy for proof obligations. We have

experimented with various relatively small algorithms and many of the proof obligations can be proved completely automatically using the automated proof strategy.

However, not all the programs can be verified completely automatically; a number of proof obligations have to be proved interactively (manually). We have analysed these problematic proof obligations, and we have discussed solutions to the problems.

The main reason why some proof obligations cannot be proved automatically is that Isabelle can not always find the required instantiations of quantifiers. The automated tactics of Isabelle use unification to find required instantiations, and we have seen that this is not effective enough for our examples. There are no other heuristics available in Isabelle to find candidates for the instantiation for quantifiers.

This is a substantial problem; quantifiers appear in the annotation of most non-trivial parallel programs.

Chapter 5

Verifying various small algorithms

We have experimented with the automated verification of the annotation of various relatively small algorithms. In this chapter we discuss the verification of these algorithms with PVS and Isabelle.

In [MW05] the verification of several relatively small algorithms is discussed, using the tool in combination with PVS. The following algorithms are discussed in [MW05]:

- monitored phase synchronization [FvG99];
- mutual exclusion using a ticket algorithm [PN02];
- mutual exclusion using Peterson's algorithm for two processes [FvG99];
- mutual exclusion using semaphores [PN02];
- parallel linear search [FvG99];
- wait-free consensus protocol [Moo02];
- wait-free handshake register [Hes98].

We have used the same fully-annotated algorithms for our experiments. While most algorithms are relatively small, the "Wait-free handshake register" is a somewhat larger algorithm, and a mechanical proof of the annotation of the algorithm is discussed in [Hes98], where the NQTHM theorem prover is used. The annotation of this algorithm consists of a large number of system invariants.

The translation of the original PVS models of the programs and their annotation from [MW05] to Isabelle was mostly straightforward. But, some of the algorithms differ slightly when modelled in Isabelle. In PVS, types may be empty, but types in Isabelle are non-empty by definition: every type in Isabelle has at least one instance. In most of the algorithms a parallel composition over a type is used, and in some cases this type may be empty, which can be modelled in PVS, but cannot be modelled in Isabelle. This is not a significant restriction, parallel programs without any components are not very useful.

We have also experimented with two other algorithms from [FvG99]: "A Simple Election Protocol" and "The Initialization Protocol". We have discussed these in detail in Chapter 3.

Table 5.1 gives an overview of the results of the experiments. *Proof script A* (see Section 4.4.2) was used for the automated Isabelle proofs and the proof script of [MW05] was used for the automated PVS proofs. *Proof script A* was used because it can prove slightly more than *Proof script B* (see Section 4.4.2) can, for our set of algorithms.

The Isabelle verification is discussed in detail in Section 4.6 and Section 4.7. Some of the algorithms cannot be verified completely automatically with Isabelle, these cases are marked bold in the table. Proof obligations that cannot be proved automatically require an interactive proof. These problematic proof obligations are discussed in Section 4.7. They have one and the same problem: Isabelle cannot automatically find the required instantiations of the quantifiers. After the manual instantiation of the quantifiers, the rest of their proof is automated.

The algorithms discussed in this chapter can all be verified automatically with the PVS proof script. Next a generalisation of one of the problematic proof obligations of Isabelle is discussed to illustrate why these proof obligations can be proved automatically with the PVS proof script. Consider the following goal (lemma):

$$\langle\, \forall d\colon\ x(d) \leq m \,\rangle \ \wedge\ \langle\, \forall d\colon\ m \leq x(d) \,\rangle \quad \Rightarrow \quad x(c0) \leq x(d0)$$

This goal is valid because if the first quantifier is instantiated with *c0* , and the second quantifier is instantiated with *d0* , then we have $x(c0) \leq m \leq x(d0)$ in the antecedent, and thus $x(c0) \leq x(d0)$ in the consequent because $\leq$ is transitive. Isabelle cannot prove this goal automatically, because unification cannot find the required instantiations. Next we discuss the strategy of the PVS proof script. PVS uses pattern matching of *subterms* of quantified formulas with *subterms* of formulas in the goal, to find candidates for the instantiation. First, the PVS proof script instantiates the quantifiers with the first candidate found, which is *c0* :

$$x(c0) \leq m \ \wedge\ m \leq x(c0) \quad \Rightarrow \quad x(\mathrm{c0}) \leq x(\mathrm{d0})$$

this goal is unprovable and backtracking is used. Next, the proof script instantiates the quantifiers with every possible candidate found (eager instantiation), which are *c0* and *d0* . This results in the following goal:

$$x(c0) \leq m \ \wedge\ m \leq x(c0) \ \wedge\ x(d0) \leq m \ \wedge\ m \leq x(d0) \quad \Rightarrow \quad x(c0) \leq x(d0)$$

This goal can be proved by PVS using the remainder of the proof script. This goal *after* instantiation can also be proved by the Isabelle proof script.

For interactive proofs, most time is spent by the user thinking about the construction of the proof, and the runtime of the proof is then rather insignificant. The runtime of automated proofs is usually longer than the runtime of interactive proofs, but the amount of required human effort decreases dramatically. The last two columns of the table show the runtime of the automated PVS and Isabelle proofs. This is the time it takes to replay the complete proof of the algorithms. Manual proofs are used for the proof obligations that could not be proved completely automatically with Isabelle, and the automated proof attempts that fail are not included in the runtime. An Intel Xeon 2.80 Ghz CPU is used for the verification. The computer has 4GB of memory, but only a small fragment of this is actually used and required. The default settings of PVS and Isabelle are used.

In general, PVS is faster than Isabelle, at least for the algorithms discussed here. There are however a few exceptions. Some of the Isabelle verifications of the very small algorithms are faster than the corresponding PVS verifications. Also, the Isabelle verifications of two of the algorithms that require manual interaction are faster than the corresponding automated PVS verifications. Manual proofs are usually more efficient than automated proofs. The PVS verification of the "Wait-free handshake register" [Hes98] is much faster than the Isabelle verification of this same algorithm.

Evaluation

The most significant problem we have encountered with the automated verification with Isabelle is how Isabelle deals with quantifiers; Isabelle can often not find the required instantiations automatically. We have not encountered this problem with PVS; various automatic instantiation strategies are available in PVS, and they have turned out to be sufficiently effective for the experiments. We have experimented with only a small set of algorithms, but quantifications appear in most non-trivial parallel programs, so it is a significant problem.

The automated proof scripts for PVS and Isabelle do not support induction. Induction is very difficult to automate, and the automated tactics of PVS and Isabelle do not support induction. We did not need induction for the verification of the algorithms discussed in this chapter. As noted in [MW05], recursion is often not required as it is often encoded in a repetition.

Algorithm	*Proof Oblig .*	*Auto*		*Runtime (sec)*	
		PVS	Isabelle	PVS	Isabelle
Simple Election Protocol (coarse-grained)	27	27	27	3	5
Simple Election protocol (finer-grained)	60	60	60	5	5
Initialization protocol for two components	216	216	216	12	18
Initialization protocol for two components (Variant 1)	676	676	676	73	124
Initialization protocol for two components (Variant 2)	75	75	75	7	12
Initialization protocol for two components (Variant 3)	75	75	75	7	9
Initialization protocol for three components	1278	1278	1278	118	321
Initialization protocol for three components (Variant 1)	2619	2619	2619	294	2143
Initialization protocol for three components (Variant 2)	236	236	236	75	154
Initialization protocol for three components (Variant 3)	236	236	236	71	118
Monitored phase synchronization	93	93	**69**	55	11
Mutual exclusion using a ticket algorithm	71	71	**59**	28	12
Mutual exclusion using Peterson's algorithm for two processes	51	51	51	9	4
Mutual exclusion using semaphores	13	13	13	2	1
Parallel linear search	17	17	17	3	1
Wait-free consensus protocol	26	26	26	2	1
Wait-free handshake register	178	178	**172**	138	1664

Table 5.1: Overview of the results of the experiments

The runtime of the PVS verifications is in general shorter than the runtime of the Isabelle verifications. Even for these reasonably small algorithms we can see that, depending on the algorithm, the difference can be significant. The difference is most significant for "Wait-free handshake register". Even though a part the Isabelle verification is done manually, it is still more than 12 times slower than the fully automated PVS verification of this algorithm. The Isabelle proof script "Proof script B" reduces the runtime of our Isabelle verification of this algorithm considerably, but is sill 4 times slower than the fully automated PVS proof. See Section 4.7.4 for more details.

Chapter 6

Proofs with universal and existential quantifications

6.1 Introduction

One of the main problems we have encountered with our automated Isabelle proofs, is that required instantiations of quantifiers are often not found. This chapter gives an overview of the available options for proofs with universal and existential quantification in PVS and Isabelle.

Tactics (or strategies) transform proof states into logically equivalent or stronger proof states. We discuss which tactics are available in which situations, and we show how various transformations of the proof states are justified logically, and why others are not allowed.

6.2 PVS

A proof state in PVS consists of zero or more subgoals to be proved. Each subgoal is a sequent formula of the form:

$$A_1\,,\,\ldots\,,\,A_n\;\vdash\;C_1\,,\,\ldots\,,\,C_m$$

where $0 \leq n$ and $0 \leq m$. $A_1 \ldots A_n$ are formulas called antecedents and $C_1 \ldots C_m$ are formulas called consequents. The logical interpretation of the sequent is:

$$A_1\;\wedge\;\ldots\;\wedge\;A_n\quad\Rightarrow\quad C_1\;\vee\;\ldots\;\vee\;C_m$$

Subgoals in the proof state are independent of each other, i.e. they do not share bound variables. This allows us to treat the subgoals in isolation. The subsections that follow discuss the options available when dealing with existential and universal quantification in PVS proofs.

6.2.1 Universal quantification in the consequents

There are two ways to deal with a universal quantifier ($\forall$) in the consequents: apply induction, or replace the bound variable with a skolem variable, which is treated as a constant in the sequent. Different types of induction are available, depending on the type of the bound variable.

In the remainder of this section we discuss the effect of skolemization on universal quantifiers in the consequents and we show how this is justified logically. Consider a sequent (goal) of the following form:

$$A(s!1)\quad\vdash\quad \texttt{FORALL}\;(t\colon T)\colon\;P(t,s!1)\;\;,\;\;C(s!1)$$

where the skolem variable $s!1$ is of type S, and S and T are arbitrary types. A skolem variable is a variable that is universally bound at the top-level of the sequent; the antecedents and the consequents of the sequent are within the scope of the universal quantifier. To distinguish skolem

variables from other variables, it is common to add an exclamation mark and an index to the name of the variable that is replaced. Many PVS tactics use this convention by default. We adopt this notation in this report: $s!1$ is a skolem variable, i.e. a constant in this sequent.

The *skolem* tactic can eliminate the quantifier from the goal by replacing it with a skolem variable ($t!1$). We show why this is allowed:

$$
\begin{array}{ll}
 & A(s!1) \quad \vdash \quad \texttt{FORALL}\ (t\colon T)\colon\ P(t, s!1)\ \ ,\ \ C(s!1) \\
\equiv & \qquad \{\ \text{definition of } \vdash\text{, translated}\ \} \\
 & \langle\ \forall(s!1\colon S)\colon \quad A(s!1) \ \Rightarrow\ \langle\ \forall(t\colon T)\colon\ P(t, s!1)\ \rangle\ \vee\ C(s!1)\ \rangle \\
\equiv & \qquad \{\ \vee \text{ distributes over } \forall\ \} \\
 & \langle\ \forall(s!1\colon S)\colon \quad A(s!1) \ \Rightarrow\ \langle\ \forall(t\colon T)\colon\ P(t, s!1)\ \vee\ C(s!1)\ \rangle\ \rangle \\
\equiv & \qquad \{\ \Rightarrow \text{ distributes over } \forall\text{, rename t into t!1}\ \} \\
 & \langle\ \forall(s!1\colon S,\ t!1\colon T)\colon \quad A(s!1) \ \Rightarrow\ P(t!1, s!1)\ \vee\ C_1(s!1)\ \rangle \\
\equiv & \qquad \{\ \text{definition of } \vdash\text{, translated back}\ \} \\
 & A(s!1) \quad \vdash \quad P(t!1, s!1)\ \ ,\ \ C(s!1)
\end{array}
$$

From this we conclude that universal quantifiers can be eliminated from the consequents by replacing their bound variable with a skolem variable. Because both the original goal and the goal after *skolem* are logically equivalent, it is still possible to transform the resulting goal back to the original. The *generalize* tactic does just that.

6.2.2 Existential quantification in the antecedents

There is only one practical way to deal with an existential quantifier ($\exists$) in the antecedents: applying skolemization. An existential quantifier in the antecedents is very similar to a universal quantifier in the consequents, and can also be replaced by a skolem variable. The following illustrates this:

$$
\begin{array}{ll}
 & \texttt{EXISTS}\ (t\colon T)\colon\ P(t, s!1)\ \ ,\ \ A(s!1) \quad \vdash \quad C(s!1) \\
\equiv & \qquad \{\ \text{Trading}\ \} \\
 & A(s!1) \quad \vdash \quad C(s!1)\ \ ,\ \ \texttt{NOT}(\texttt{EXISTS}\ (t\colon T)\colon\ P(t, s!1)) \\
\equiv & \qquad \{\ \text{De Morgan}\ \} \\
 & A(s!1) \quad \vdash \quad C(s!1)\ \ ,\ \ \texttt{FORALL}\ (t\colon T)\colon\ \texttt{NOT}(P(t, s!1)) \\
\equiv & \qquad \{\ \text{See Section 6.2.1: t becomes a skolem variable t!1}\ \} \\
 & A(s!1) \quad \vdash \quad C(s!1)\ ,\ \texttt{NOT}(P(t!1, s!1)) \\
\equiv & \qquad \{\ \text{Trading}\ \} \\
 & P(t!1, s!1)\ ,\ A(s!1) \quad \vdash \quad C(s!1)
\end{array}
$$

where *Trading* denotes:

$$P\ ,\ Q\ \vdash\ R \quad \equiv \quad Q\ \vdash\ R\ ,\ \neg P$$

From this we conclude that existential quantifiers can be eliminated from the antecedents by replacing their bound variables with skolem variables. The resulting goal is logically equivalent to the original. The *skolem* tactic can be used for this transformation.

Remark. PVS automatically moves negated formulas in the antecedents to the consequents, and moves negated formulas in the consequents to the antecedents. This way the antecedent and consequent formulas never have a negation at the top level. So, some of the intermediate sequents in the derivation above are never presented to the user in this form. They are automatically normalised to a form without negation on the top level. The right hand side of the *Trading* equivalence is always automatically translated to the left hand side by PVS.

Induction cannot be applied directly to existential quantifiers in the antecedents. However, induction can be applied indirectly by first replacing the quantifier with a skolem variable, and then using the *generalize* tactic. This results in a universal quantifier that binds the variable, in the consequent.

6.2.3 Universal quantification in the antecedents

There are no tactics available to replace a universal quantifier in the antecedents with a skolem variable. The reason for this is that this is only possible if the type of the bound variable is non-empty, and even in these situations it is hardly ever useful as it generally strengthens the goal. The only practical way to use a universal quantifier in the antecedents is to instantiate the quantifier.

In the remainder of this section we first discuss why, and in which situations, a skolem variable could, *in theory*, be replaced by a skolem variable, and finally we discuss how quantifiers can be instantiated.

Consider a sequent (goal) of the following form:

$$\texttt{FORALL}\ (t\colon T)\colon P(t, s!1)\ \ ,\ \ A(s!1)\ \ \vdash\ \ C(s!1))$$

where skolem variable $s!1$ is of type S, and S and T are arbitrary types. We have seen in previous sections that the quantifier can *in theory* be replaced by a skolem variable if we manage to move the quantifier to the top level of the goal. This is only possible if $\wedge$ distributes over the universal quantifier, which is not the case if T can be empty, as this could weaken the goal.

If T is non-empty, i.e. $\exists(t : T) : \mathit{true}$, then $\wedge$ does distribute over the universal quantifier. In that case, the quantifier could, *in theory*, be replaced by a skolem variable:

$$\begin{array}{ll}
 & \texttt{FORALL}\ (t\colon T)\colon P(t, s!1)\ \ ,\ \ A(s!1)\ \ \vdash\ \ C(s!1) \\
\equiv & \quad \{\ \text{Definition of } \vdash\text{, translated}\ \} \\
 & \langle\ \forall(s!1\colon S)\colon\ \ \langle\ \forall(t\colon T)\colon\ P(t, s!1)\ \rangle\ \wedge\ A(s!1)\ \ \Rightarrow\ \ C(s!1)\ \rangle \\
\equiv & \quad \{\ \text{Because } \langle\ \exists(t\colon T)\colon\ \mathit{true}\ \rangle\text{, } \wedge \text{ distributes over } \forall\} \\
 & \langle\ \forall(s!1\colon S)\colon\ \ \langle\ \forall(t\colon T)\colon\ P(t, s!1)\ \wedge\ A(s!1)\rangle\ \ \Rightarrow\ \ C(s!1)\ \rangle \\
\equiv & \quad \{\langle\ \exists(t\colon T)\colon\ \mathit{true}\ \rangle\ \ ,\ \ [\langle\ \forall(t\colon T)\colon\ P(t)\ \rangle\ \Rightarrow\ Q\ \ \equiv\ \ \langle\ \exists(t : T)\colon\ P(t) \Rightarrow Q\ \rangle\ ]\ \} \\
 & \langle\ \forall(s!1\colon S)\colon\ \ \langle\ \exists(t\colon T)\colon\ P(t, s!1)\ \wedge\ A(s!1)\ \ \Rightarrow\ \ C(s!1)\ \rangle\ \rangle \\
\Leftarrow & \quad \{\langle\ \exists(t\colon T)\colon\ \mathit{true}\ \rangle\ \ ,\ \ [\langle\ \forall(t\colon T)\colon\ P(t)\ \rangle\ \ \Rightarrow\ \ \langle\ \exists(t\colon T)\colon\ P(t)\ \rangle]\ \ , \\
 & \qquad \text{rename } t \text{ to } t!1\ \} \\
 & \langle\ \forall(s!1\colon S,\ t!1\colon T)\colon\ \ P(t!1, s!1)\ \wedge\ A(s!1)\ \ \Rightarrow\ \ C(s!1)\ \rangle \\
\equiv & \quad \{\ \text{Definition of } \vdash\text{, translated back}\ \} \\
 & P(t!1)\ \ ,\ \ A(s!1)\ \ \vdash\ \ C(s!1)
\end{array}$$

The universal quantifier in the antecedents is effectively weakened to an existential quantifier (see Section 6.2.2), and weakening an antecedent formula strengthens the goal. This strengthening of the goal most likely renders the goal unprovable, and is therefore not very practical. There are no tactics available to do this transformation directly, but can be done manually.

We do not have much choice but to instantiate the quantifier, possibly after copying the assumption first to ensure that the goal is not strengthened. We can either instantiate the quantifier ourselves, with a term of the correct type, or use the *inst?* tactic (or a tactic that uses *inst?* indirectly) and let PVS automatically choose terms to instantiate the quantifier with. PVS matches *subterms* of quantified formulas with *subterms* of other formulas in the goal in order to find candidates for instantiation. The most important automatic instantiation strategies are:

1. Use the first candidate found (`inst? :if-match first`).
2. Use the candidate that generates the fewest TCCs (`inst? :if-match best`).
3. Instantiate all quantifiers at the top level of the formula, using all candidates found (`inst? :if-match all`). I.e., if applied to $\langle\ \forall x, y\colon Q(x, y)\ \rangle$, then both x and y are instantiated, and only subterms that contain instantiations for both bound variables are considered to be candidates.

There is also an option that limits automatic instantiations to those that do not result in TCCs. This option can be used in combination with each of the strategies above.

6.2.4 Existential quantification in the consequents

An existential quantifier in the consequents in similar to a universal quantifier in the antecedents:

$$
\begin{array}{ll}
 & A(s!1) \quad \vdash \quad \texttt{EXISTS}\ (t\colon T)\colon\ P(t, s!1) \quad , \quad C(s!1) \\
\equiv & \qquad \{\ \text{Trading}\ \} \\
 & A(s!1) \quad , \quad \texttt{NOT}(\texttt{EXISTS}\ (t\colon T)\colon\ P(t, s!1)) \quad \vdash \quad C(s!1) \\
\equiv & \qquad \{\ \text{De Morgan}\ \} \\
 & A(s!1) \quad , \quad \texttt{FORALL}\ (t\colon T)\colon\ \texttt{NOT}(P(t, s!1)) \quad \vdash \quad C(s!1)
\end{array}
$$

Exactly the same options as those for the universal quantification in the antecedents are available.

6.3 Isabelle

A proof state in Isabelle consists of zero or more subgoals. A subgoal is presented as:

$$\bigwedge p_1 \ldots p_m .\quad [\![\, A_1 \,;\, \ldots \,;\, A_n \,]\!] \quad \Longrightarrow \quad C$$

where $0 \leq m$ and $0 \leq n$. $\bigwedge$ is omitted if $m = 0$ (no bound variables), both brackets are omitted if $n = 1$ (only one assumption), and brackets and arrow are omitted if $n = 0$ (no assumptions). $p_1 \ldots p_m$ are (universally) bound variables at Isabelle's meta-level. They are local to a goal and they are treated as constants. $A_1 \ldots A_n$ are assumptions and C is the conclusion. The goal is logically interpreted as:

$$\langle\ \forall(p_1\,,\,\ldots\,,\,p_m)\colon\ \ A_1\ \wedge\ \ldots\ \wedge\ A_n\ \Rightarrow\ C\ \rangle$$

where bound variables $p_1 \ldots p_m$ can occur in $A_1 \ldots A_n$ and C. Schematic variables (or unknown variables or meta variables) are variables defined at the meta-level, they are placeholders and they can be replaced by arbitrary terms of the correct type that do not depend on the bound variables (including $p_1 \ldots p_m$). The names of schematic variables start with a question mark. Logically, a schematic variable is a variable existentially bound at the top level of the proof state; subgoals share schematic variables. Consider for example the following proof state:

1. $\bigwedge p .\quad [\![\, A_1(p) \,;\, A_2(p) \,]\!] \quad \Longrightarrow \quad C_1(p, ?x)$
2. $A_3(?x) \quad \Longrightarrow \quad C_2$

The proof state consists of two subgoals, and they share the schematic variable $?x$. The logical interpretation of this proof state is:

$$\left\langle\ \exists x\colon\quad \underbrace{\langle\ \forall p\colon\ A_1(p)\ \wedge\ A_2(p)\quad \Rightarrow\quad C_1(p,x)\ \rangle}_{\text{subgoal 1}}\quad \wedge\quad \underbrace{(A_3(x)\ \Rightarrow\ C_2)}_{\text{subgoal 2}}\ \right\rangle$$

The subsections that follow discuss the possibilities of using universal quantifiers, existential quantifiers and schematic variables in Isabelle proofs.

Remark. Schematic variables are defined at the meta-level and their logical interpretation depends on where they appear:

- Schematic variables in subgoals of the proof state are logically equivalent to existential quantification at the top level of the proof state, as we have seen in this section.
- When a lemma is proved and stored, the free variables in the lemma are replaced by schematic variables. Consider for example the theorem $?P \implies ?P \vee ?Q$ (the *disjI1* introduction rule). When this theorem is used in a proof, the schematic variables can be instantiated and the resulting theorem appears in the assumptions of a goal. The proved theorem should be interpreted as the fact: $\langle\ \forall P, Q\colon\ P\ \Rightarrow\ P\ \vee\ Q\ \rangle$.

– Schematic variables can be used in the body of a theorem to prove, for example: `lemma X: "1 + 2 = ?P"`. This states that we are trying to prove that there exists a term that is logically equivalent to $1 + 2$. When the proof is completed, the witness that is found is substituted for `?P`; the schematic variable does not appear in the theorem that is stored. In this case terms like 3 or $1 + 2$ can be substituted for `?P`. If the first is used, the stored theorem (with the name `X`) states the fact: `1 + 2 = 3`.

6.3.1 Universal quantification in the conclusion

There is only one practical way to deal with a universal quantifier ($\forall$) in the conclusion, and that is replacing the quantifier with a bound variable at the meta-level:

$$
\begin{array}{ll}
 & \bigwedge p.\ \ A(p) \implies \text{ALL } x.\ C(p,x) \\
\equiv & \quad \{ \text{ Translated } \} \\
 & \langle\ \forall p:\ \ A(p) \Rightarrow \langle\ \forall x:\ C(p,x)\ \rangle\ \rangle \\
\equiv & \quad \{ \Rightarrow \text{distributes over } \forall\ \} \\
 & \langle\ \forall p, x:\ \ A(p) \Rightarrow C(p,x)\ \rangle \\
\equiv & \quad \{ \text{ Translate back to Isabelle notation } \} \\
 & \bigwedge p\ x.\ \ A(p) \implies C(p,x)
\end{array}
$$

The introduction rule *allI* can be used to do this transformation: `apply(rule allI)`. PVS goals and Isabelle goals are represented differently. Because of this difference, the goal discussed here is slightly different from the discussed goal for PVS in Section 6.2.1. Also, we did not need to specify the types of the variables in Isabelle. Isabelle supports polymorphic typing, similar to Haskell and ML, and is able to infer the most general type of each term.

Induction cannot directly be applied to a universal quantifier in the conclusion. Induction can only be applied to bound variables that appear only in the conclusion: the universal quantifier must first be transformed to a bound variable at the meta-level (as described above).

6.3.2 Existential quantification in the assumptions

An existential quantifier in the assumptions is very similar to a universal quantifier in the conclusion, and the only practical way to deal with an existential quantifier in the assumptions is to replace it with a bound variable at the meta-level:

$$
\begin{array}{ll}
 & \bigwedge p.\ \ [\![\, (\text{EX } x.\ A_1(p,x))\ ;\ A_2(p)\,]\!] \implies C(p) \\
\equiv & \quad \{ \text{ Translated } \} \\
 & \langle\ \forall p:\ \ \langle\ \exists x: A_1(p,x)\ \rangle\ \wedge\ A_2(p) \ \ \Rightarrow\ \ C(p)\ \rangle \\
\equiv & \quad \{ \text{ Trading, De Morgan } \} \\
 & \langle\ \forall p:\ \ A_2(p) \ \ \Rightarrow\ \ C(p) \vee \langle\ \forall x:\ \neg A_1(p,x)\ \rangle\ \rangle \\
\equiv & \quad \{ \vee \text{ distributes over } \forall\ \} \\
 & \langle\ \forall p:\ \ A_2(p) \ \ \Rightarrow\ \ \langle\ \forall x:\ C(p) \vee \neg A_1(p,x)\ \rangle\ \rangle \\
\equiv & \quad \{ \Rightarrow \text{distributes over } \forall\ \} \\
 & \langle\ \forall p, x:\ \ A_2(p) \ \ \Rightarrow\ \ C(p) \vee \neg A_1(p,x)\ \rangle \\
\equiv & \quad \{ \text{ Trading } \} \\
 & \langle\ \forall p, x:\ \ A_1(p,x) \wedge A_2(p) \ \ \Rightarrow\ \ C(p)\ \rangle \\
\equiv & \quad \{ \text{ Translate back to Isabelle notation } \} \\
 & \bigwedge p\ x.\ \ [\![\, A_1(p,x)\ ;\ A_2(p)\,]\!] \implies C(p)
\end{array}
$$

The elimination rule *exE* can be used for this transformation: `apply(erule exE)`.

6.3.3 Universal quantification in the assumptions

In an interactive proof the user usually instantiates the universal quantifiers in the assumptions manually, using either the elimination rule *allE* or the destruction rule *spec*. Instantiation of a universal quantifier in the assumptions is allowed because it weakens the assumptions and thus strengthens the goal. The Automated tactics do not instantiate quantifiers directly, they replace the quantifier with a schematic variable and use unification to find candidates for the instantiation of the schematic variable. We use a proof state with two subgoals to show the effect of the transformation on a proof state and we show its logical interpretation:

$$
\begin{array}{ll}
 & \mathbf{1.} \quad \bigwedge p.\ \ \text{ALL } x.\ A(p,x) \implies C_1(p) \\
 & \mathbf{2.} \quad C_2 \\
\equiv & \quad \{ \text{ Translated } \} \\
 & \langle\ \forall p\colon\ \langle\ \forall x\colon A(p,x)\ \rangle\ \Rightarrow\ C_1(p)\ \rangle\ \wedge\ C_2 \\
\equiv & \quad \{ \text{ contraposition, De Morgan } \} \\
 & \langle\ \forall p\colon\ \neg C_1(p)\ \Rightarrow\ \langle\ \exists x\colon \neg A(p,x)\ \rangle\ \rangle\ \wedge\ C_2 \\
\equiv & \quad \{ \text{ Types in Isabelle are non-empty, } \vee \text{ distributes over } \exists, \text{ definition } \Rightarrow \} \\
 & \langle\ \forall p\colon\ \langle\ \exists x\colon\ A(p,x)\ \Rightarrow\ C_1(p)\ \rangle\ \rangle\ \wedge\ C_2 \\
\Leftarrow & \quad \{ \star\ \ \langle\ \forall x\colon \langle\ \exists y\colon P(x,y)\ \rangle\ \rangle\ \Leftarrow\ \langle\ \exists z\colon \langle \forall x\colon P(x,z(x))\ \rangle\ \rangle\ \} \\
 & \langle\ \exists x\colon\ \langle\ \forall p\colon\ A(p,x(p))\ \Rightarrow\ C_1(p)\ \rangle\ \rangle\ \wedge\ C_2 \\
\equiv & \quad \{ \wedge \text{ distributes over } \exists\ \} \\
 & \langle\ \exists x\colon\ \langle\ \forall p\colon\ A(p,x(p))\ \Rightarrow\ C_1(p)\ \rangle\ \wedge\ C_2\ \rangle \\
\equiv & \quad \{ \text{ Translate back to Isabelle notation } \} \\
 & \mathbf{1.} \quad \bigwedge p.\ \ A(p,?x(p)) \implies C_1(p) \\
 & \mathbf{2.} \quad C_2
\end{array}
$$

And the proof of $\star$ using Isabelle's proof style:

$$
\begin{array}{ll}
 & \langle\ \forall x\colon \langle\ \exists y\colon P(x,y)\ \rangle\ \rangle\ \Leftarrow\ \langle\ \exists z\colon \langle \forall x\colon P(x,z(x))\ \rangle\ \rangle \\
\equiv & \quad \{ \text{ Definition } \Longrightarrow \} \\
 & \langle\ \exists z\colon \langle \forall x\colon P(x,z(x))\ \rangle\ \rangle\ \Longrightarrow\ \langle\ \forall x\colon \langle\ \exists y\colon P(x,y)\ \rangle\ \rangle \\
\equiv & \quad \{ \text{ Eliminate } \exists \text{ from assumption (see Section 6.3.2) } \} \\
 & \bigwedge z.\ \ \langle \forall x\colon P(x,z(x))\ \rangle\ \Longrightarrow\ \langle\ \forall x\colon \langle\ \exists y\colon P(x,y)\ \rangle\ \rangle \\
\equiv & \quad \{ \text{ eliminate } \forall \text{ from conclusion (see Section 6.3.1), rename x to } x' \ \} \\
 & \bigwedge z\ x'.\ \ \langle \forall x\colon P(x,z(x))\ \rangle\ \Longrightarrow\ \langle \exists y\colon P(x',y)\ \rangle \\
\equiv & \quad \{ \text{ Instantiate } \forall \text{ in assumption with } x', \text{ instantiate } \exists \text{ in assumption with } z(x') \ \} \\
 & \bigwedge z\ x'.\ \ P(x',z(x'))\ \Longrightarrow\ P(x',z(x')) \\
\equiv & \quad \{ \text{ Trivial } \} \\
 & \mathit{true}
\end{array}
$$

If we accept the validity of the (somewhat controversial) axiom of choice, the result is actually logically equivalent to the original. However, in Isabelle the *allE* elimination rule – which is used for the translation above – is defined as an unsafe rule. It is strongly advised to apply *safe* rules before applying *unsafe* rules. Even if a goal is still provable after the application of this unsafe rule *allE*, the proof may now require an application of the axiom of choice (or a related axiom), which is not used by the automated tactics. In Section 6.3.5 is discussed how schematic variables can be used.

Universal quantifications in the assumptions are also used by the simplifier. The simplifier uses universal quantifiers in the assumptions to create rewrite rules. If the simplifier is applied to a goal with assumption $\langle\ \forall x\colon f(x)\ \rangle$, then the rewrite rule $f(?x) = \mathit{true}$ is applied to the goal and conclusions such as $f(x1)$ can be proved automatically.

6.3.4 Existential quantification in the conclusion

An existential quantifier in the conclusion is very similar to a universal quantifier in the premises; the same options are available. The existential quantifier can again be replaced by a schematic

variable:

$$
\begin{array}{ll}
 & \bigwedge p.\quad A(p) \implies \text{EX } x.\ C(p,x) \\
\equiv & \quad \{ \text{ Trading, De Morgan } \} \\
 & \bigwedge p.\quad [\![A(p)\ ;\ \text{ALL } x.\ \neg C(p,x)]\!] \implies \mathit{False} \\
\Leftarrow & \quad \{ \text{ Universal quantification is removed from the assumptions: see section 6.3.3 } \} \\
 & \bigwedge p.\quad [\![A(p)\ ;\ \neg C(p,?x(p))]\!] \implies \mathit{False} \\
\equiv & \quad \{ \text{ trading } \} \\
 & \bigwedge p.\quad A(p) \implies C(p,?x(p))
\end{array}
$$

The *exI* introduction rule can be used for this transformation.

6.3.5 Using schematic variables

Variables bound by quantifiers are often translated to schematic variables. In this section we discuss how schematic variables can be used in proofs. We illustrate the possibilities using examples.

The first example shows how schematic variables can be instantiated, discusses one of the consequences of sharing schematic variables among subgoals, and illustrates the importance of backtracking:

$$
\begin{array}{ll}
 & \text{EX } (x::\mathit{int}).\ f(x) \wedge f(x-1) \implies \text{EX } y.\ f(y) \wedge f(y+1) \\
\equiv & \quad \{ \text{ Translated, omitting type information } \} \\
 & \langle\, \exists x:\ f(x) \wedge f(x-1) \,\rangle \Rightarrow \langle\, \exists y:\ f(y) \wedge f(y+1) \,\rangle \\
\equiv & \quad \{ \text{ Eliminate existential quantifier from the assumptions, see Section 6.3.2 } \} \\
 & \langle\ \forall x:\ f(x) \wedge f(x-1) \Rightarrow \langle\, \exists y:\ f(y) \wedge f(y+1) \,\rangle\ \rangle \\
\Leftarrow & \quad \{ \text{ Eliminate existential quantifier from the conclusion, see Section 6.3.4 } \} \\
 & \langle\, \exists y: \langle\ \forall x:\ f(x) \wedge f(x-1) \Rightarrow f(y(x)) \wedge f(y(x)+1)\ \rangle \,\rangle \\
\equiv & \quad \{ \text{ Eliminating } \wedge \text{ in conclusion, tactic: } \texttt{apply(rule conjI)}, \text{ this yields two} \\
 & \quad\ \text{subgoals } \} \\
 & \langle\, \exists y: \\
 & \quad \langle\, \forall x:\ f(x) \wedge f(x-1) \Rightarrow f(y(x)) \,\rangle \quad \wedge \\
 & \quad \langle\, \forall z:\ f(z) \wedge f(z-1) \Rightarrow f(y(z)+1) \,\rangle \\
 & \,\rangle \\
\Leftarrow & \quad \{ \text{ Generalise/instantiate schematic variable: } y := \lambda x.\ x-1 \text{ , } \beta\text{-reduction } \} \\
 & \quad \langle\, \forall x:\ f(x) \wedge f(x-1) \Rightarrow f(x-1) \,\rangle \\
 & \wedge \quad \langle\, \forall z:\ f(z) \wedge f(z-1) \Rightarrow f(z-1+1) \,\rangle \\
\equiv & \quad \{ \text{ Trivially } \mathit{true}, \text{ proved using the simplifier } \} \\
 & \mathit{true}
\end{array}
$$

Schematic variables are usually not instantiated manually in interactive proofs; the user usually instantiates the existential or universal quantification directly, without transforming it to a schematic variable first, or lets Isabelle find them. Schematic variables can however be instantiated manually using the *instantiate_tac* tactic.

Instantiation for the schematic variable can often be found automatically using unification of the conclusion with the assumptions, using the *assumption* tactic, which is applied to a single subgoal. Before the instantiation of the schematic variable in the proof above, the first application of *assumption* to the first subgoal (the first conjunct) yields the instantiation $\lambda x.\ x$. Using this instantiation the first subgoal can be proved (the first conjunct is equal to *true*) but the second subgoal (second conjunct) cannot be proved using this instantiation. When backtracking (the `back` command) is used at this point, then the second candidate for the instantiation is used, which is $\lambda x.\ x-1$. Using this instantiation, both goals can be proved, as we have seen in the proof above. Application of *assumption* to the second subgoal before instantiation fails; no candidates can be found in the second subgoal.

Unification is the only mechanism available in Isabelle to automatically find instantiations for schematic variables. If unification fails, then we have no other choice but to instantiate the schematic variable manually, or better: instantiate the original quantifier directly.

In [NPW02, Section 5.11] an example of an unsuccessful proof is given that illustrates why schematic variables cannot be instantiated with terms that depend on bound variables. We discuss the same example and proof, and present the goals in (predicate) logic, as we are used to. Note that the goal is not provable, not without additional information about f.

$$
\begin{array}{ll}
 & \text{ALL } x.\ f(x,x) \implies \text{EX } y.\ \text{ALL } z.\ f(y,z) \\
\equiv & \quad \{ \text{ Translated } \} \\
 & \langle\, \forall x\colon f(x,x)\,\rangle \Rightarrow \langle\, \exists y\colon \langle\, \forall z\colon\ f(y,z)\,\rangle\,\rangle \\
\Leftarrow & \quad \{ \text{ Eliminate existential quantifier from the conclusion, see Section 6.3.4 } \} \\
 & \langle\, \exists y\colon\ \langle\, \forall x\colon f(x,x)\,\rangle \Rightarrow \langle\, \forall z\colon\ f(y,z)\,\rangle\,\rangle \\
\equiv & \quad \{ \text{ Eliminate universal quantifier from the conclusion, see Section 6.3.1 } \} \\
 & \langle\, \exists y\colon \langle\, \forall z\colon\ \langle\, \forall x\colon f(x,x)\,\rangle \Rightarrow f(y,z)\ \rangle\,\rangle \\
\Leftarrow & \quad \{ \text{ Eliminate universal quantifier from the assumption, see Section 6.3.3 } \} \\
 & \langle\, \exists x,y\colon \langle\, \forall z\colon\ f(x(z),x(z)) \Rightarrow f(y,z)\ \rangle\,\rangle \\
\equiv & \quad \{ \text{ Translate back to Isabelle notation } \} \\
 & \bigwedge z.\ \ f(?x(z),?x(z)) \implies f(?y,z)
\end{array}
$$

The goal can only be proved if $?y$ can be instantiated with z, but this is obviously not possible as it is bound at a deeper level of the formula. Compare this with the following goal and (successful) proof:

$$
\begin{array}{ll}
 & \text{ALL } x.\ f(x,x) \implies \text{ALL } z.\ \text{EX } y.\ f(y,z) \\
\equiv & \quad \{ \text{ Translated } \} \\
 & \langle\, \forall x\colon f(x,x)\,\rangle \Rightarrow \langle\, \forall z\colon \langle\, \exists y\colon\ f(y,z)\,\rangle\,\rangle \\
\equiv & \quad \{ \text{ Eliminate universal quantifier from the conclusion, see Section 6.3.1 } \} \\
 & \langle\, \forall z\colon\ \langle\, \forall x\colon f(x,x)\,\rangle \Rightarrow \langle\, \exists y\colon\ f(y,z)\,\rangle\,\rangle \\
\Leftarrow & \quad \{ \text{ Eliminate universal quantifier from the assumption, see Section 6.3.3 } \} \\
 & \langle\, \exists x\colon \langle\, \forall z\colon\ f(x(z),x(z)) \Rightarrow \langle\, \exists y\colon\ f(y,z)\,\rangle\,\rangle\,\rangle \\
\Leftarrow & \quad \{ \text{ Eliminate existential quantifier from the conclusion, see Section 6.3.4 } \} \\
 & \langle\, \exists x,y\colon \langle\, \forall z\colon\ f(x(z),x(z)) \Rightarrow f(y(z),z)\,\rangle\,\rangle \\
\Leftarrow & \quad \{ \text{ Instantiate variables: } x := \lambda z.\ z \text{ , } y := \lambda z.\ z \text{ , } \beta\text{-reduction } \} \\
 & \langle\, \forall z\colon\ f(z,z) \Rightarrow f(z,z)\,\rangle \\
\equiv & \quad \{ \text{ Trivial } \} \\
 & \mathit{true}
\end{array}
$$

The instantiations of the schematic variables are found by the *assumption* tactic: $f(?x(z),?x(z))$ unifies with $f(?y(z),z)$ with $\sigma = \{?x \mapsto \lambda z.\ z\ ,\ ?y \mapsto \lambda z.\ z\}$.

6.4 Conclusions

PVS and Isabelle handle universal quantifications in the consequent/conclusion and existential quantification in the antecedent/assumptions in a similar way: induction can be applied (directly or indirectly) or the quantifier can be eliminated by replacing the bound variable with a parameter (or skolem variable) that is local to the goal.

However, PVS and Isabelle use a different approach to dealing with universal quantifiers in the antecedents/assumptions and to existential quantifiers in the consequents/conclusion. In interactive proofs with PVS and Isabelle, the user usually instantiates the quantifiers with the desired terms, but in automated proofs we rely on the facilities of PVS and Isabelle to provide the instantiations that are needed.

PVS uses pattern matching of *subterms* of the quantified formula with *subterms* of the antecedent and consequent formulas, to find candidate instantiations. The user can choose between a number of automatic instantiation strategies to be used, ranging from instantiation with the first candidate found to instantiation with every possible candidate found (eager instantiation). The quantified formula is usually removed after is has been instantiated, this can strengthen the goal; extra candidates may appear later in the proof, after the quantifier has already been instantiated and removed. So, the decision at which point in the proof process a quantifier is instantiated, can have a considerable influence on the success of finding the appropriate instantiations. The user can choose which of the instantiation strategies is to be used by the automated tactics.

In Isabelle, the automated tactics do not instantiate the quantifiers directly: the quantifiers are translated to schematic variables, which delays the instantiation. It is important that safe rules are applied first (especially the *allI* and *exE* rule) before the quantifiers are replaced by schematic variables. This is because bound variables at the meta level are not available for the instantiation of the schematic variable (see Section 6.3.5). Only the bound variables that have been introduced before introducing the schematic variable are available through parameters of the schematic variables. Isabelle uses unification of the conclusion with the assumptions to find candidates for the instantiation of the schematic variables. If the conclusion can be unified with an assumption, then the goal can be proved by using the candidate found by unification. Unfortunately, this may render the subgoals that share the schematic variable unprovable. When unification finds more than one possible candidate, then one is chosen and backtracking is available to use the other candidates, if the goals cannot be proved using the first candidate. The automated tactics use unification in combination with backtracking.

Chapter 7

Verification of a Garbage Collection Algorithm

7.1 Introduction

In this chapter the verification of the annotation of a larger algorithm is discussed: a garbage collection algorithm from [PN02].

In [PN02], the annotation of this garbage collection algorithm is mechanically proved in Isabelle, using her formalisation of Owicki/Gries in Isabelle. This formalisation is also discussed in [PN02]. We have proved the same annotation using the tool in combination with PVS. The results of our verification are compared with the results of the original in [PN02].

Our verification serves two main purposes: First, to explore the limits of PVS and the standard proof script of [MW05]. Second, to make a comparison between PVS and Isabelle. The amount of human effort required to prove the correctness of the annotation with the theorem provers is compared. Also, the specification of the algorithm and its annotation in Isabelle and PVS is compared.

Problem description

A memory location is called garbage if it has been allocated by an application but can never be accessed again by this application. The *collector* is responsible for identifying and collecting memory locations that are garbage, and making them available again. This memory location can then be claimed by applications. The *mutator* represents such an application that uses memory and can 'produce' garbage. [PN02] discusses a single-mutator variant and a multi-mutator variant. We discuss the single-mutator variant.

The specification of the collector is as follows:

do *true* →
  Identify garbage
  {x is garbage}
  Append x to the list of free memory
od

The task of the collector is to continuously search for locations that are not accessible, identify the garbage and then append this garbage to the list of free memory. The property that is verified is a safety property, and states that every location that is appended to the list of free memory is indeed garbage.

7.2 Model

In this section the model of the algorithm, and its formalisation in PVS are discussed. We do not focus on the design of the algorithm, nor on the design decisions. For a detailed description of the algorithm, the annotation and the correctness of the annotation, we refer to [PN02]. We focus on the automated verification of the annotation using the tool.

7.2.1 Memory

The memory is modelled as a finite directed graph with a fixed number of nodes and a fixed number of edges. A predetermined, non-empty, subset of nodes called *roots* is always accessible. A node is *reachable*, or accessible, if it is a root or there is at least one (directed) path from one of the roots to this node. Nodes that are not reachable are garbage.

Every node has a colour: *black* or *white*. The type *color* is defined accordingly; it has exactly two elements and is defined as an alias for *bool*, where *true* represents *black* and *false* represents *white*.

PVS

```
black: bool = true
white: bool = false
color: TYPE = bool

nEdges: posnat
nNodes: posnat

nodeIdx: TYPE = below(nNodes)
edgeIdx: TYPE = below(nEdges)

nodes: TYPE = [ nodeIdx -> color ]
edge: TYPE = [# src: nodeIdx,
                dest: nodeIdx
             #]

edges: TYPE = [ edgeIdx -> edge ]
Roots: (nonempty?[nodeIdx])

Reach(e: edges): setof[nodeIdx] =
    {i:nodeIdx |
         Roots(i) OR
          EXISTS (path: [nat -> edgeIdx]):
            EXISTS (r: nodeIdx): Roots(r) AND
              EXISTS (len: nat): len > 0 AND
                e(path(0))‘src = r AND
                e(path(len-1))‘dest = i AND
                FORALL (j: nat): j < len - 1 IMPLIES
                   e(path(j))‘dest = e(path(j+1))‘src
    }
```

The function *BtoW* expresses whether an edge points from a black node to a white node and *Blacks* returns the set of black nodes. The memory is in a safe state if all *reachable* nodes are black. The function *Safe* expresses whether a state is safe.

PVS

```
BtoW (e:edge)(m:nodes): bool =
   m(e‘src) = black AND m(e‘dest) = white

Blacks(m:nodes): setof[nodeIdx] =
   {i: nodeIdx | m(i) = black}
```

PVS

```
Safe(m:nodes, e:edges): bool =
   FORALL (x: nodeIdx): Reach(e)(x) IMPLIES Blacks(m)(x)
```

7.2.2 Algorithm

A (record) type *state* is defined to denote a state of the system:

PVS

```
state: TYPE = [#   M: nodes,
                   E: edges,
                   bc: setof[nodeIdx],
                   obc: setof[nodeIdx],
                   Ma: nodes,
                   ind: nat,
                   k: nodeIdx,
                   z: bool
              #]
```

The *mutator* component and the *collector* component are executed in parallel.

```
0:    {z} {T ∈ Reach(E)} {inv: Roots ≠ ∅}
      co
       proc
1:        {z} {T ∈ Reach(E)}
          Mutator-Component
4:
       corp
       proc
5:
          Collector-Component
37:
       corp
      oc
```

Even though it is already encoded in the type that *root* is a non-empty set, PVS does not always automatically use this fact in proofs. We need a system invariant to state this fact explicitly. T is an arbitrary node and R is an arbitrary edge. Both are defined as constants.

PVS

```
R: edgeIdx
T: nodeIdx
```

Mutator component The *mutator* component redirects the edge R to the reachable node T and then colours node T black. This is repeated forever.

```
Mutator-Component =
1:    {z} {T ∈ Reach(E)}
      do true →
2:        {z} {T ∈ Reach(E)}
          dest(E(R)), z := T, ¬z
3:      ;{¬z} {T ∈ Reach(E)}
          M(T), z := black, ¬z
      od
```

Collector-Component =

5: **do** *true* →
  Blacken-roots
12: ;{*Roots* ⊆ *Blacks*(M)}
  obc := ∅
13: ;{*Roots* ⊆ *Blacks*(M)} {obc = ∅}
  bc := *Roots*
14: ;{*Roots* ⊆ *Blacks*(M)} {obc = ∅} {bc = *Roots*}
  Ma := λ*n*. *if* *n* ∈ *Roots* *then* black *else* white
15: ;{*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(Ma)}{*Blacks*(Ma) ⊆ bc}
    {bc ⊆ *Blacks*(M)} {obc ⊂ *Blacks*(Ma) ∨ *Safe*(M, E)}
  do obc ≠ bc →
16: {*Roots* ⊆ *Blacks*(M)} {bc ⊆ *Blacks*(M)}
    obc := bc
    Propagate-Black
23: ;{*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)} {bc ⊆ *Blacks*(M)}
      {obc ⊂ *Blacks*(M) ∨ *Safe*(M, E)}
    Ma := M
24: ;{*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(Ma)} {*Blacks*(Ma) ⊆ *Blacks*(M)}
      {bc ⊆ *Blacks*(M)} {obc ⊂ *Blacks*(Ma) ∨ *Safe*(M, E)}
    bc := ∅
25: *Count-Blacks*
  od
31: *Append*
od

Collector component The *collector* component consists of four different parts. The first part is labelled *blacken roots*, in this part the roots are coloured black. Observe that the roots are reachable by definition. The second part is labelled *propagate black*. During this phase every white node n, for which there is an edge e from a black node to this node, is coloured black. The third part is labelled *Count blacks*. In this part the component determines whether it should stop colouring, this is when no new nodes have been coloured black during the last iteration. The last part is labelled *append*, at this point we are certain that all reachable nodes are black, and the white nodes are not reachable and thus garbage. Note that because of the possible interference of the mutator, it is not guaranteed that all black nodes are reachable. The garbage is appended to the list of free memory.

Append_to_free is used to add nodes to the free list. Analogous to [PN02], it is defined as an uninterpreted function, and an axiom defines the necessary property.

PVS

```
Append_to_free: [nodeIdx -> [edges -> edges]]
append_to_free_def: AXIOM
   FORALL (n: nodeIdx, e: edges, n2: nodeIdx):
     NOT(Reach(e)(n)) IMPLIES
       (Reach(Append_to_free(n)(e))(n2)) = (n2 = n OR Reach(e)(n2))
```

At control point 35 we see how the original specification is translated: the statement at control point 35 is the only place in the algorithm where a node is added to the list of free nodes, and the pre-condition {ind < nNodes ∧ ind ∉ *Reach*(E)} guarantees that this node is not reachable, and thus garbage.

7.2.3 Partial definitions

[PN02] uses lists to represent the collection of nodes and edges. Variables of type *nat* are used to refer to nodes and edges in the lists. The lists are finite and the length of the lists (which is

Blacken-roots =

6: ind := 0
7: ;{ind ≤ nNodes ∧ *BRInv*(ind)}
do ind < nNodes →
8: {ind < nNodes ∧ *BRInv*(ind)}
if ind ∈ *Roots* →
9: {ind < nNodes ∧ *BRInv*(ind)} {ind ∈ *Roots*}
M(ind) := black
[] ind ∉ *Roots* →
10: {ind < nNodes ∧ *BRInv*(ind)} {ind ∉ *Roots*}
skip
fi
11: ;{ind < nNodes ∧ *BRInv*(ind + 1)}
ind := ind + 1
od
12: {*Roots* ⊆ *Blacks*(M)}

BRInv ≡ λ*idx*. ({ *i* | *i* < *idx* } ∩ *Roots*) ⊆ *Blacks*(M)

Propagate-Black =

17: {*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)} {bc ⊆ *Blacks*(M)}
ind := 0
18: ;{*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)}
{bc ⊆ *Blacks*(M)} {*PBInv*(ind)} {ind ≤ nEdges}
do ind < nEdges →
19: {*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)}
{bc ⊆ *Blacks*(M)} {*PBInv*(ind)} {ind < nEdges}
if M(*src*(E(ind))) = black →
20: {*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)}
{bc ⊆ *Blacks*(M)} {*PBInv*(ind)} {ind < nEdges ∧ M(*src*(E(ind))) = black}
k := *dest*(E(ind))
21: ;{*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)} {bc ⊆ *Blacks*(M)}
{*PBInv*(ind)} {ind < nEdges ∧ M(*src*(E(ind))) = black} {*Auxk*}
M(k), ind := black, ind + 1
[] M(*src*(E(ind))) ≠ black →
22: {*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)}
{bc ⊆ *Blacks*(M)} {*PBInv*(ind)} {ind < nEdges}
⟨ **if** M(*src*(E(ind))) = black →
skip
[] M(*src*(E(ind))) ≠ black →
ind := ind + 1
fi ⟩
fi
od
23: {*Roots* ⊆ *Blacks*(M)} {obc ⊆ *Blacks*(M)}
{bc ⊆ *Blacks*(M)} {obc ⊂ *Blacks*(M) ∨ *Safe*(M, E)}

PBInv ≡ λ*idx*. obc ⊂ *Blacks*(*M*) ∨
⟨ ∀*i* : *i* < *idx* ⇒ ¬*BtoW*(E(*i*), M) ∨
(¬z ∧ *i* = R ∧ *dest*(E(R)) = T ∧
⟨ ∃*r* : *idx* ≤ *r* ∧ *BtoW*(E(*r*), M) ⟩) ⟩

Auxk ≡ M(k) ≠ black ∨ ¬*BtoW*(ind, M) ∨ obc ⊂ *Blacks*(M) ∨
(¬z ∧ ind = R ∧ *dest*(E(R)) = T ∧ ⟨ ∃*r* : ind < *r* ∧ *BtoW*(E(*r*), M) ⟩)

Count-Blacks =

25: $\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subseteq \mathit{Blacks}(\mathrm{Ma})\}\ \{\mathit{Blacks}(\mathrm{Ma}) \subseteq \mathit{Blacks}(\mathrm{M})\}$
$\{\mathrm{bc} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subset \mathit{Blacks}(\mathrm{Ma}) \ \vee\ \mathit{Safe}(\mathrm{M},\mathrm{E})\}\ \{\mathrm{bc} = \emptyset\}$
ind := 0

26: ;$\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subseteq \mathit{Blacks}(\mathrm{Ma})\}\ \{\mathit{Blacks}(\mathrm{Ma}) \subseteq \mathit{Blacks}(\mathrm{M})\}$
$\{\mathrm{bc} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subset \mathit{Blacks}(\mathrm{Ma}) \ \vee\ \mathit{Safe}(\mathrm{M},\mathrm{E})\}\ \{\mathit{CountInv}(\mathrm{ind})\}\ \{\mathrm{ind} \leq \mathrm{nNodes}\}$
do ind < nNodes →

27: $\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subseteq \mathit{Blacks}(\mathrm{Ma})\}$
$\{\mathit{Blacks}(\mathrm{Ma}) \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{bc} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subset \mathit{Blacks}(\mathrm{Ma}) \ \vee\ \mathit{Safe}(\mathrm{M},\mathrm{E})\}$
$\{\mathit{CountInv}(\mathrm{ind})\}\ \{\mathrm{ind} < \mathrm{nNodes}\}$
if M(ind) = black →

28: $\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subseteq \mathit{Blacks}(\mathrm{Ma})\}$
$\{\mathit{Blacks}(\mathrm{Ma}) \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{bc} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subset \mathit{Blacks}(\mathrm{Ma}) \ \vee\ \mathit{Safe}(\mathrm{M},\mathrm{E})\}$
$\{\mathit{CountInv}(\mathrm{ind})\}\ \{\mathrm{ind} < \mathrm{nNodes} \wedge \mathrm{M(ind)} = \mathrm{black}\}$
bc := bc ∪ {ind}

[] M(ind) ≠ black →

29: $\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subseteq \mathit{Blacks}(\mathrm{Ma})\}$
$\{\mathit{Blacks}(\mathrm{Ma}) \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{bc} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subset \mathit{Blacks}(\mathrm{Ma}) \ \vee\ \mathit{Safe}(\mathrm{M},\mathrm{E})\}$
$\{\mathit{CountInv}(\mathrm{ind}+1)\}\ \{\mathrm{ind} < \mathrm{nNodes}\}$
skip

fi

30: ;$\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subseteq \mathit{Blacks}(\mathrm{Ma})\}$
$\{\mathit{Blacks}(\mathrm{Ma}) \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{bc} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathrm{obc} \subset \mathit{Blacks}(\mathrm{Ma}) \ \vee\ \mathit{Safe}(\mathrm{M},\mathrm{E})\}$
$\{\mathit{CountInv}(\mathrm{ind}+1)\}\ \{\mathrm{ind} < \mathrm{nNodes}\}$
ind := ind + 1

od

CountInv ≡ $\lambda \mathit{idx}.\ (\{\, i \mid i < \mathit{idx} \,\} \cap \mathit{Blacks}(\mathrm{Ma})) \subseteq \mathrm{bc}$

Append =

31: $\{\mathit{Roots} \subseteq \mathit{Blacks}(\mathrm{M})\}\ \{\mathit{Safe}(\mathrm{M},\mathrm{E})\}$
ind := 0

32: ;$\{\mathit{AppendInv}(\mathrm{ind})\}\ \{\mathrm{ind} \leq \mathrm{nNodes}\}$
do ind < nNodes →

33: $\{\mathit{AppendInv}(\mathrm{ind})\}\ \{\mathrm{ind} < \mathrm{nNodes}\}$
if M(ind) = black →

34: $\{\mathit{AppendInv}(\mathrm{ind})\}\ \{\mathrm{ind} < \mathrm{nNodes} \wedge \mathrm{M(ind)} = \mathrm{black}\}$
M(ind) := white

[] M(ind) ≠ black →

35: $\{\mathit{AppendInv}(\mathrm{ind})\}\ \{\mathrm{ind} < \mathrm{nNodes} \wedge \mathrm{ind} \notin \mathit{Reach}(\mathrm{E}))\}$
E := *Append_to_free*(ind, E)

fi

36: ;$\{\mathit{AppendInv}(\mathrm{ind}+1)\}\ \{\mathrm{ind} < \mathrm{nNodes}\}$
ind := ind + 1

od

AppendInv ≡ $\lambda \mathit{idx}.\ (\{\, i \mid \mathit{idx} \leq i \,\} \cap \mathit{Reach}(\mathrm{E})) \subseteq \mathit{Blacks}(\mathrm{M})$

constant) has to be stated explicitly at every point in the annotation.

In PVS predicate subtypes are available to create subtypes of existing types by including only the elements of an existing type that satisfy a certain predicate. We declare the types *nodeIdx* and *edgeIdx* to be subsets of the naturals. For example we have defined the type *nodeIdx* to contain only the natural numbers below *nNodes*:

PVS
```
nodeIdx: TYPE = {i:nat | i < nNodes}
```

Instead of lists, we use functions over *nodeIdx* and *edgeIdx* to represent the collection of nodes and edges, respectively. The collector iterates over the nodes and edges in a fixed order (ascending). As is often the case, the loop terminates when the loop variable is outside a range; this variable can therefore not be of type *nodeIdx* or *edgeIdx*. Like [PN02], we use a loop variable of type *nat* to iterate over the nodes and edges; this introduces a number of complications.

Type checking in PVS is often not decidable, TCCs (Type Correctness Condition) are generated to force the user to prove that the types of the terms are correct. Most TCCs are trivial and are automatically proved by PVS. Everywhere in the specification where a term t of type *nat* is used where a term of a subtype of *nat* (i.e. *nodeIdx*) is expected, a TCC is generated that forces us to prove that the term is a member of the subset (i.e. $t < nNodes$).

In the remainder of this section we discuss how the PVS model of the algorithm and the model of its annotation is influenced by the decision to use predicate subtypes.

Statements

Because the *weakest liberal preconditions* are modelled in PVS in isolation from the model of the annotation, there are some complications. Assume the *wlp* of the statement M(ind) := *black* at control point 9 is modelled as:

PVS
```
wlp_stat_9(p: predicate): predicate =
    lambda (s:state): p (s WITH [`M(s`ind) := black])
```

The type of $s`ind$ is *nat* and the type of $s`M$ is $nodeIdx \Rightarrow color$. The typechecker generates the following TCC for this fragment:

PVS
```
wlp_stat_9_TCC1: OBLIGATION FORALL (s: state): s`ind < nNodes
```

We are forced to prove that every natural number is smaller than *nNodes*, which is obviously not true. There are three possible solutions to this problem: The first and most obvious solution is to not use predicate subtypes for the model. The second solution is quite drastic, we could model the proof obligations differently. By combining the model of the *wlp* and the model of the relevant annotation, it can be concluded that $ind < nNodes$ is true, as it is one of the pre-conditions of that statement. The third option is to change the definition and the PVS model of the *wlp* of the statement.

Predicate subtypes increase the readability, and generated TCCs that cannot be proved automatically are a sign of inconsistencies and errors, detected at an early stage. Combining the definitions would be a very drastic change and the definition of *wlp* and annotation are decoupled for a reason: it facilitates incremental verification. The third solution is preferred.

We introduce a notion of *partial statements*. A partial statement is an atomic statement S, accompanied by a condition C. The partial statement states that S is executed, but that no guarantees can be given regarding the state after executing the statement S, *unless* it is

guaranteed that the state before executing statement S satisfies condition C. The *weakest liberal precondition* of a partial statement $S!C$, with post-condition Q is:

$$
\begin{array}{ll}
 & wlp.(S!C).Q \\
\equiv & \quad \{definition\ wlp\ partial\ statement\} \\
 & [C \Rightarrow wlp.S.Q \quad \wedge \quad \neg C \Rightarrow false] \\
\equiv & \quad \{Definition \Rightarrow and \neg\} \\
 & [C \Rightarrow wlp.S.Q \quad \wedge \quad C] \\
\equiv & \quad \{Modus\ ponens\} \\
 & [C \wedge wlp.S.Q]
\end{array}
$$

Each statement S that is only defined for states that satisfy C is changed to the partial statement $S!C$, where C is the TCC generated by PVS. Applied to the example, the statement at control point 9 is changed to:

$$(\text{ind} := \text{black})!(\text{ind} < \text{nNodes})$$

and the new model of the *wlp* becomes:

PVS
```
wlp_stat_9(p: predicate): predicate =
    LAMBDA (s:state):
      (s‘ind < nNodes) AND p (s WITH [‘M(s‘ind) := black])
```

The following TCC is generated by PVS:

PVS
```
wlp_stat_9_TCC1: OBLIGATION FORALL (s: state):
                      s‘ind < nNodes IMPLIES s‘ind < nNodes
```

This resulting TCC is trivially true and is proved automatically by PVS.

Remark. The type checker appears to evaluate the conjunction in the formula (*wlp_stat_9*) from left to right. Even though conjunction is commutative (logically), if the positions of the two conjuncts are interchanged, then we are still presented with the original, invalid TCC.

An alternative model of the *wlp* of the partial statement is:

PVS
```
wlp_stat_9(p: predicate): predicate =
    LAMBDA (s: {x:state | x‘ind < nNodes}):
      p (s WITH [‘M(s‘ind) := black])
```

Using this model of the *wlp* of the partial statement, the type of the function is still $state \Rightarrow bool$, but when the function is applied to a state s in a proof, an extra subgoal is created that forces us to prove that $s‘ind < nNodes$ (the condition C) is true in the context of the lemma/proof obligation.

Guards

The if-statements used by [PN02] are different from the if-statements used by the tool. An if-statement of [PN02] behaves like a selection statement used by most popular programming languages. The if-statement used by the tool is a blocking statement, the component blocks until one of the guards evaluates to *true*. The if-statement of [PN02] is of the form:

if G **then** $\{P\}\ S_1$
else $\{Q\}\ S_2$
fi

where the else clause is optional. This is translated to the if-statement used by the tool:

$$\begin{array}{lrcl} \textbf{if} & G & \rightarrow & \{P\}\ S_1 \\ [] & \neg G & \rightarrow & \{Q\}\ S_2 \\ \textbf{fi} & & & \end{array}$$

Because either G or $\neg G$ evaluates to *true*, the statement cannot possibly block, thereby progress is guaranteed. Some guards G are only defined for states that satisfy a condition C and unprovable TCCs are generated for their models. In such cases we strengthen guard G with the required condition C into $C \wedge G$. For the second guard the negation $\neg(C \wedge G)$ is then used. Because $C \wedge G$ is defined, the second guard is defined. The second guard is logically equivalent to:

$$\begin{array}{ll} & \neg(C \wedge G) \\ \equiv & \quad \{\ \text{De Morgan}\ \} \\ & \neg C \vee \neg G \\ \equiv & \quad \{\ \text{Definition}\ \Rightarrow\ \} \\ & C \Rightarrow \neg G \end{array}$$

In these cases C is implied by the pre-conditions of the statement, in fact: for each guard, the conjunction of the pre-conditions of the statement and the original guard are logically equivalent to the conjunction of the same pre-conditions and the new guard. Therefore the correctness of the annotation is not influenced by this transformation.

Assertions

Usually we split conjunctive assertions into co-assertions, one for each conjunct, and prove the correctness of the assertions separately. This transformation is valid because of the following equivalence

$$\{P\}S\{Q \wedge R\} \quad \equiv \quad \{P\}S\{Q\} \wedge \{P\}S\{R\}$$

The conjunctive assertion cannot be split this way if R is only defined for states that satisfy Q because the assertions are modelled separately from their co-assertions. For example the assertion

$$\{\text{ind} < \text{nNodes} \wedge \text{M(ind)} = \text{black}\}$$

at control point 34 cannot be split this way because *M(ind)* is only defined when $ind < nNodes$. They can however be split using

$$\{P\}S\{Q \wedge R\} \quad \equiv \quad \{P\}S\{Q\} \wedge \{P\}S\{Q \Rightarrow R\}$$

We don't split such assertions if R is only defined when Q is true; we only split the conjunctive assertions if the conjuncts are defined in isolation.

7.3 Proving the correctness of the annotation

We have worked with two slightly different models of the algorithm.

Model 1: Parallel composition over elements of a type Initially, the tool did not support binary parallel composition and we had to translate the binary parallel composition as used in [PN02] to a parallel composition over the elements of a type (see Section 3.3.1 for more information).

Model 2: Binary parallel composition Later, after support for binary parallel composition had been added to the tool, we created a model using this operator. The resulting model is more like the original model in [PN02] and this is the model discussed in this report.

The only difference between the proof obligations generated for the first model and those generated for the second model is that a large number of extra proof obligations are required for the proof of the first. They result from the translation of the parallel construct, and their proof is trivial. All other proof obligations are comparable, they are in fact almost identical.

Initially, a considerable amount of effort was required to prove the proof obligations. We have investigated why this has been the case, and our second attempt to prove the proof obligations automatically has turned out to be more effective.

7.3.1 First attempt

The first model has been used for our first attempt. 59 out of the total of 3972 proof obligations generated for this model could not be proved automatically, at least not within the 30 minutes we were willing to wait for each proof obligation. 9 of them could be proved using proof hints and 50 proof obligations required a manual proof.

Five non-trivial lemmas about graphs and the *Reach* function were used for the manual proofs. These lemmas require manual proofs, which is not surprising as their proofs require induction and clever case analysis.

```
Graph1: LEMMA
        FORALL (t: nodeIdx):
          FORALL (e: edges):
            FORALL (r: edgeIdx):
               Reach(e)(t) IMPLIES
                  Reach(e WITH [‘r‘dest := t])(t)

Graph2: LEMMA
        FORALL (M: nodes):
          FORALL (E:edges):
            (FORALL (e: edgeIdx):
               NOT(BtoW(E(e))(M))) AND subset?(Roots,Blacks(M))
              IMPLIES
                 Safe(M,E)

Graph3: LEMMA
         FORALL (E: edges, r: nodeIdx):
            Reach(E)(r) IMPLIES
              FORALL (n: nodeIdx, e: edgeIdx):
                 Reach(E WITH [(e)‘dest := r])(n) IMPLIES
                    Reach(E)(n)

Graph4: LEMMA
       FORALL (E: edges, M: nodes, index: {i: nat | i <= nEdges}):
          (  Reach(E)(T) AND
             subset?(Roots,Blacks(M)) AND
             (FORALL (i:edgeIdx): i < index IMPLIES NOT(BtoW(E(i))(M))) AND
              R < index AND
              M(E(R)‘src) = black AND
              M(T) /= black
           ) IMPLIES
                EXISTS (r: edgeIdx):
                  index <= r AND BtoW((E WITH [‘R‘dest := T])(r))(M)
```

```
Graph5: LEMMA
       FORALL (E: edges, M: nodes):
              (  Reach(E)(T) AND
                 subset(Roots,Blacks(M)) AND
                 (FORALL (i:edgeIdx): i < R IMPLIES NOT(BtoW(E(i))(M))) AND
                 M(E(R)‘src) = black AND
                 M(E(R)‘dest) = black AND
                 M(T) /= black
              ) IMPLIES
                  EXISTS (r: edgeIdx):
                     R < r AND BtoW((E WITH [‘R‘dest := T])(r))(M)
```

In addition to the five non-trivial lemmas about graphs, the proof of the problematic proof obligations still required extensive manual interaction. Most noticeable, because PVS constantly tried to use definitions in superfluous assumptions, and the proof script attempted to instantiate large numbers of quantifiers with a large number of terms, the proof state often became too complex. The proof script typically did not terminate within the set time. In these situations we had to resort to manual proofs and had to constantly remove assumptions that were not required for the proof, and we had to instantiate quantifiers manually.

The proof of five proof obligations needed the *append_to_free_def* axiom; they had to be proved manually for this reason. The *append_to_free_def* axiom was used as a rewrite rule for these proofs.

7.3.2 Second attempt

The second model has been used for the second attempt. 533 proof obligations are generated for this model.

Reach function We observe that every single proof obligation with a proof that requires the definition of the *Reach* function had to be proved manually. To prevent the automatic expansion of the *Reach* function we define *Reach* as an uninterpreted function and specify the definition with an axiom:

```
Reach: [edges -> setof[nodeIdx]]

Reach_def: AXIOM
  FORALL (e:edges)(i: nodeIdx):
    Reach(e)(i) =
    (     Roots(i) OR
          EXISTS (path: [nat -> edgeIdx]):
            EXISTS (r: nodeIdx): Roots(r) AND
              EXISTS (len: nat): len > 0 AND
                e(path(0))‘src = r AND
                e(path(len-1))‘dest = i AND
                FORALL (j: nat): j < len - 1 IMPLIES
                   e(path(j))‘dest = e(path(j+1))‘src
    )
```

If this new definition has any influence on our automatic proofs, then it can only be a positive influence because we already have to prove each proof obligation with a proof that requires the definition of the *Reach* function manually. Not only can more proof obligations be proved automatically using this specification of *Reach*, the runtime of the proof of many other proof obligations is also reduced considerably. There are indeed no proof obligations that could be proved automatically using the old definition, but could not be proved automatically using the new specification.

Instantiation of quantifiers over predicate subtypes The proof scripts use PVS' capabilities of finding instantiations for quantifiers. Quantifiers over a predicate subtype of a supertype

T are instantiated with terms of type T and terms of (predicate) subtypes of T. TCCs are generated to prove that the types are compatible. Consider for example the following goal:

```
A1(t,v)
FORALL (x:V): A2(x)
├──────────────────
C
```

where t is of type T, v is of type V, and V is a subtype of T that contains the elements of T that satisfy a predicate P, i.e. $V = \{t{:}T \mid P(t)\}$. The application of `inst? :if-match all` to this goal results in the following two subgoals:

```
A1(t,v)
A2(t)
A2(v)
├──────────
C
```

TCC:

```
A1(t,v)
A2(v)
├──────────
C
P(t)
```

The left subgoal is the original subgoal after instantiating the quantifier with the terms t and v. The right subgoal is the TCC that is generated for the instantiation with t. It has to be proved that either t is indeed an element of V (i.e. $P(t)$) or that C can be proved without using the instantiation with t.

We make extensive use of predicate subtypes and many quantifiers are instantiated with many terms with incompatible types. This results in many goals that are essentially equivalent ($P(t)$ cannot be proved) that have many assumptions and consequents. The goals become so large that the automated tactics cannot prove the goal. The *bddsimp* tactic, which is used by all the automated tactics, has to be manually interrupted in these cases because it does not terminate when applied to such large goals, at least not within the 30 minutes we were willing to wait.

Even though it is not documented in the *PVS Prover Guide* [SORSC01], the tactic `inst?`, which is used as instantiator, has the option to allow or disallow automatic instantiations that yield TCCs that do not simplify to *true* (see also suggestion 293 on the PVS suggestion list [SUG]). We change the main part of the default proof script to disallow instantiations that yield TCCs that cannot be proved automatically by simplification:

```
(branch (grind :if-match nil)
     ((then (try (reduce) (fail) (skip))
          (then (inst? :if-match all :tcc? nil) (then
                    (reduce :if-match all :instantiator m_inst?$) (fail) ) ) )) )
```

We have created a PVS strategy (LISP function) *m_inst*, which is a wrapper function for the instantiator strategy *inst?*. The only difference between the two is that *m_inst?* by default disallows instantiations that yield TCCs that do not simplify *true*. We need this strategy because *reduce* does not have an option to control this directly, we can however supply an alternative instantiator function:

```
(defstep m_inst? (&optional (fnums *) subst (where *)
                            copy? if-match polarity? (tcc? nil))
               (inst? fnums subst where :copy? copy? :if-match if-match
                            :polarity? polarity? :tcc? tcc?)
""
"")
```

This function belongs in the file *pvs-strategies* in the working directory, which is automatically loaded by PVS.

This new proof script is successful for the proof obligations of this algorithm because every instantiation that yields a nontrivial TCC that cannot be proved by the simplifier, is unnecessary and unwanted. This may very well not be the case for other algorithms.

After the changes there are 29 proof obligations left that cannot be proved automatically. One of them can be proved using proof hints and the proof of each the remaining 28 requires either one of the five lemmas about graphs or the *append_to_free_def* rewrite rule. In contrary to the first attempt, the proof of the 29 proof obligations do not require extensive manual interaction, they can all be proved by only introducing the required lemmas, and limiting the assumptions being used, in combination with the standard proof script. The amount of effort required for the proof is reduced considerably by these changes.

The runtime of the proofs is roughly about ten minutes for each attempt, if the large number of extra (trivial) proof obligations generated for the first model are disregarded.

7.3.3 Experiences manual proofs

Quantifier instantiation The strategy `inst? :if-match all` is used to instantiate quantifiers with multiple terms of the correct type. Based on the manual, our first impression was that all terms of the correct type would be considered as candidate for the instantiations, but, this is not in line with our experiences.

What happens is the following. To find candidate terms for the instantiation, PVS (pattern) matches subterms of quantified formulas with subterms of other formulas in the sequent. Each successful match delivers candidates for the bound variables of the quantified formula. Only the subterms that contain *all* variables bound at the most outer level of the formula are used. Consider for example the quantified formula

```
FORALL (x:X): FORALL (y:Y): P(x) OR Q(y)
```

The term `P(x) OR Q(y)` is the only term that is matched with subterms of formulas in the sequent. This term matches with, for example, `P(x1) OR Q(y1)`, but does not match with `P(x1)` or `Q(y1) OR P(x1)`, in another formula. If candidate instantiations are found for a quantifier, then the quantifier is instantiated with every candidate and the original quantified formula is removed. If for all quantified formulas not a single candidate is found, then PVS tries option: `if-match nil`. Using this option, even subterms that contain only some of the variables bound on the outer level are matched with subterms of formulas in the sequent. If there is a subterm `P(x1)` in the goal, then `x1` is a candidate for instantiation. The quantifiers are instantiated with the *first* candidate found, before being removed from the goal.

The behaviour of `if-match all` is often desired, and is successful for the automated proof of the proof obligations of this algorithm. Using pattern matching, instantiations can often be limited to the ones that are most likely to succeed. But for some of the manual proofs of the first attempt, there were cases in which the correct instantiations could not be found, and we had to resort to manual instantiation. Consider for example the following goal:

```
FORALL (i:T): FORALL (j:T): F(i)
F(t1)
F(t2) IMPLIES G(t2)
|-------
G(t2)
```

This goal is valid; we would like to find instantiation *t2* for the outermost quantifier. If `if-match all` is used, then no instantiation is found, since there is no term that contains both bound variables. Next, PVS tries `if-match nil`. Using this option, the term `F(i)` is matched with subterms of other formulas: matches `F(t1)` and `F(t2)` are found. The quantifier is instantiated with the first candidate: `t1`, and the quantified formula is removed. This renders the goal unprovable. In this case we prefer instantiation of every possible term with the correct type.

Other instantiation strategies are available in PVS. One of them is `if-match first*`. Based on the manual, we considered this to be a serious candidate for the default proof script: the goal above can be proved automatically using the strategy because quantifiers are instantiated with every term possible. We did some experiments with this option, and the results show that it does

not work as described in the manual [SORSC01]. We were not able to determine what it does exactly.

Extensionality Determining the equality of functions appears to be problematic for the automated proof strategies. The principle of extensionality states that two functions P and Q are equal if $\langle\, \forall x\colon P(x) = Q(x) \,\rangle$. Unfortunately, the automated proof strategies do not use this fact and we have to apply the tactic `apply-extensionality` explicitly, when extensionality is required. Consider the following goal:

$$\frac{\text{M(k)} = \text{black}}{\text{M} \quad = \quad \text{M WITH [k := black]}}$$

Even though this is trivially valid, the automated proof strategies are not able to prove this unless `apply-extensionality` is applied first.

The function `strict_subset?(P,Q)` is defined in the prelude (standard library) as:

```
strict_subset?(P, Q): bool = subset?(P, Q) & P /= Q
```

The automated tactics are in many cases not able to prove goals with this function in the consequent, as the proof of $P \neq Q$ requires extensionality. For this reason we use an alternative (but equivalent) definition:

```
subset?(P, Q) AND NOT(subset(Q, P))
```

which is logically equivalent to:

$$\langle\, \forall x\colon P(x) \Rightarrow Q(x) \,\rangle \quad \wedge \quad \langle\, \exists x\colon Q(x) \wedge \neg P(x) \,\rangle$$

Using this definition, extensionality is not required.

Last minute note In this section we have claimed that the automated tactics of PVS do not use extensionality at all. Unfortunately, this is not completely accurate. Even though this cannot be found in the PVS manuals (i.e. the prover guide [SORSC01]), we have learned from the release notes of PVS that there are versions of the automated tactics available that do apply extensionality at some points in the proof. The goal which we used as an example in this section, can be proved automatically using these tactics, but we did not carry out any further experiments and we can therefore not tell wether this would be more effective for automated proofs.

The current documentation of PVS is from november 2001, and is thus four years old. Several PVS versions have been released in these four years.

7.4 Evaluation

7.4.1 Specification

The PVS specification is very similar to the original Isabelle specification in [PN02]. There are two notable differences. First, lists are used in the original specification. The specification language of Isabelle is based on functional programming, and many functions on list are available in Isabelle, together with many theorems on lists, which can be very useful for manual proofs. In PVS not as many theorems about lists are available and the inductive character of the list datatype makes automated verification difficult; induction is not supported by the proof script and the automated tactics.

Instead of lists of elements of some type T, we use functions from a subset of the naturals to T. The functions are used like arrays; the subset of naturals represents the indices of the array, and T is the type of the elements in the array. For this algorithm, arrays can be used instead of lists because in [PN02] the lists have a constant size and their elements are accessed exclusively through their index.

Second, Isabelle does not support predicate subtyping. In [PN02] edges and nodes are of type *nat*. Extra conditions stating the acceptable ranges of the naturals are required in the annotation as there are a limited and constant number of nodes and edges. For example, a node must be a natural smaller than the size of the list representing the collection of nodes. In our specification this information is stated implicitly in the type definition. This is more readable and less error-prone.

7.4.2 Proofs

Both the original verification in [PN02] and our verification are based on the Owicki/Gries theory. The proof obligations are essentially the same for both formalisations, albeit in a slightly different form.

Isabelle proof of [PN02]

Five nontrivial lemmas are used for the verification in Isabelle. They are similar to the five nontrivial lemmas we use in PVS, and their proofs also require extensive manual interaction. The results of the verification are summarised and concluded in [PN02] as follows:

> The Owicki-Gries method splits the proof into a large number of simple interference freedom subproofs. These are very tedious to prove by hand, and so avoided by humans, who prefer to concentrate on the few difficult cases. By applying the formalized Owicki-Gries system most of the interference freedom proofs for the final annotations where carried out by Isabelle/HOL. For the remaining cases, five non-trivial lemmas about graphs had to be supplied. The proofs of these lemmas, however, were very interactive.

Reading this we got the impression that the proof was completely automated, apart from the five lemmas that had to be supplied and proved interactively. Studying the Isabelle proofs of [PN02], which are publicly available from [PNG], we discovered that this is not the case. The majority of proofs are indeed trivial and can be proved using *simp* or *force*, but still there are many that are not trivial and are long. They include, among other things, case distinction, instantiation of quantifiers and the application of rules from the standard library.

We have carried out some experiments with the Isabelle specification and proof of [PN02]. The experiments indicate that if arrays are used instead of lists, more can be proved automatically with Isabelle.

Our automated PVS proof

Our proof is almost completely automated, apart from the five nontrivial lemmas. The proof obligations that cannot be proved completely automatically using the default proof script only require the introduction of one of the five nontrivial lemmas or the *append_to_free_def* lemma, in addition to the default proof script.

Comparison

It is difficult to make a meaningful and honest comparison between our automated PVS proof and the mechanised Isabelle proof in [PN02], because of differences in the specification. Also, [PN02] does not put as much emphasis on *fully* automated verification as we do. In the remainder of this section we discuss some of our findings.

The five non-trivial lemmas about graphs, which are used for the manual proofs, are almost the same for both our verification and the verification of [PN02], which is no coincidence. The outline of the proofs are comparable, but in the Isabelle proofs case distinction is frequently used at places where in our proof the automated proof strategies can prove the goal directly.

There are goals in the Isabelle proof of [PN02] that cannot be proved automatically with Isabelle, but can be proved automatically with PVS. We have identified two reasons why they could be proved with PVS: instantiation of quantifiers and the decision procedures of PVS. We use a proof obligation to illustrate this. Consider the Hoare-triple:

$$\{\langle\, \forall i\colon\, i < n \;\Rightarrow\; I(i)\,\rangle \;\;\wedge\;\; I(n)\} \quad n \;:=\; n+1 \quad \{\langle\, \forall i\colon\, i < n \;\Rightarrow\; I(i)\,\rangle\}$$

with proof obligation:

$$\langle\, \forall i\colon\, i < n \;\Rightarrow\; I(i)\,\rangle \;\wedge\; I(n) \quad\Rightarrow\quad \langle\, \forall i\colon\, i < (n+1) \;\Rightarrow\; I(i)\,\rangle$$

where n is of type *nat*. This type of proof obligation occurs frequently in the verification of programs, particulary in situations where an invariant has to be re-established at the end of a repetition. Both in PVS and in Isabelle we first eliminate the quantifier and implication from the consequent:

$$\langle\, \forall i\colon\, i < n \;\Rightarrow\; I(i)\,\rangle \;\wedge\; I(n) \;\wedge\; i' < (n+1) \quad\Rightarrow\quad I(i')$$

where i' is a constant. This goal can be proved by instantiating the quantifier with i'

$$i' < n \;\Rightarrow\; I(i') \;\wedge\; I(n) \;\wedge\; i' < (n+1) \quad\Rightarrow\quad I(i')$$

and then use case distinction on $i' < n$, after which the goals are trivially valid. In PVS, both the instantiation and the case distinction are applied by the automated strategies. In Isabelle, the automated tactics do not find the required instantiation, and even if we instantiate the quantifier manually, the goal can still not be proved automatically. In fact, the following goal cannot be proved automatically in Isabelle:

$$[\![\, I(n)\,;\; i' \le n\,;\; n \le i' \,]\!] \quad\Longrightarrow\quad I(i')$$

but can be proved using the decision procedures of PVS.

Automatic instantiation of quantifiers is crucial for our PVS verification. In the proof of [PN02], many instantiations are not found automatically, but instantiation is often avoided in the interactive proof by using additional lemmas on sets.

7.5 Conclusions

In this chapter we have shown the verification of a larger algorithm with PVS; a large part of the verification of the algorithm is completely automated. The instantiation strategies of PVS and the proof script have proved to be very effective, but they are not perfect.

This case study has shown that for the automated verification it is crucial to choose the datatypes carefully. The choice to use predicate subtypes turned out to be a good one: not only is the specification much clearer, but after a change to the proof script we were also able to prevent many unwanted instantiations. This would not have been possible if the approach of [PN02] would have been used. Using lists instead of arrays decreases the amount of possible automation of the proofs, both in PVS and Isabelle.

This case study revealed two problems with automated verification in Isabelle: First, the needed instantiations of quantifiers are often not found; the quantifiers have to be instantiated manually. Second, many goals that can be proved automatically by using the decision procedures of PVS, cannot be proved automatically in Isabelle.

Chapter 8

Conclusions

The aim of the project was to determine and compare the effectiveness of the interactive theorem provers PVS and Isabelle with respect to automated verification of Owiki/Gries outlines. Support for Isabelle has been successfully added to the proof generator tool and we have experimented with the automated verification of a number of parallel algorithms. We have verified a number of relatively small algorithms, and a larger garbage collection algorithm. Almost 95% of the proof obligations of this garbage collection algorithm can be proved automatically with PVS. In this section the most important findings and conclusions of our experiments are discussed.

Specification Language

The specification language of the theorem provers is important as programs, annotations and proof obligations have to be modelled efficiently. [MW05] shows how the proof obligations can be modelled in PVS. The translation of the models of the proof obligations to Isabelle turned out to be straightforward.

The complexity of the models of programs and annotation varies, depending on the program being verified. We have encountered only a few problems with the modelling of the programs we experimented with.

In contrast to PVS, types in Isabelle are always non-empty. This restricts the parallel composition over the elements of a type, as used by the tool, to a non-empty set of processes when Isabelle is used for the verification.

In PVS predicate subtypes are available to create subtypes of existing types. Predicate subtypes allow natural specification of subtypes and unproved TCCs generated by the type checker are an indication of inconsistencies, detected at an early stage. There is no equivalence of predicate subtypes in Isabelle.

Automated verification

Both PVS and Isabelle can reduce the human effort required to verify parallel programs considerably. Many proof obligations can be proved completely automatically using PVS or Isabelle. The proof obligations that cannot be proved completely automatically have to be proved interactively with the theorem provers, but even those proofs can largely be automated.

Quantifiers The validity problem of formulas in higher-order predicate logic is undecidable; fully automated theorem proving is not always possible. We have found automated proving of proof obligations with quantifiers to be problematic in PVS and Isabelle. The required instantiations of quantifiers are usually obvious in interactive proofs, but it is difficult to find them automatically.

PVS and Isabelle use different strategies to find the required instantiations of quantifiers. Isabelle uses unification of the conclusion with the assumptions to find instantiations. PVS uses pattern matching of subterms of quantified formulas with other subterms in the goal, to find

candidates for the instantiation. The user can choose between the various automatic instantiation strategies of PVS, ranging from instantiation with the first candidate found to instantiation with every candidate found (eager instantiation). The advantages of Isabelle's strategy are that the instantiation is delayed as much as possible, and that only serious candidates are considered for the instantiation. But, many of the required instantiations are not found automatically.

The advantage of PVS' eager instantiation strategy is that more instantiations are found. But, many of the instantiations are unnecessary and complicate the goal. Another disadvantage is that the instantiation of the quantifiers strengthens the goal because the original quantified formula is removed; this may render the goal unprovable.

We have found PVS' instantiation strategies to be more effective than the strategy of Isabelle; more proof obligations can be proved completely automatically. The relatively small programs we have experimented with can all be verified completely automatically with PVS in combination with the generic proof script. Some of the programs can however not be verified completely automatically with Isabelle because the needed instantiations for quantifiers are not always found. This is a significant problem, as the annotation of most non-trivial parallel programs contain quantifiers.

Decision procedures The comparison of the mechanised Isabelle verification of the garbage collection algorithm of [PN02] with our automated PVS verification of the same algorithm revealed that the decision procedures of PVS succeed at proving certain goals that cannot be proved automatically with Isabelle. More specifically, we have found the PVS decision procedures concerning naturals to be more effective than those of Isabelle. These proofs require extra human interaction in Isabelle, such as case distinction.

Types The choice of types used for the specification of programs and their annotation can have a significant influence on the efficiency of the automated verification. Because automated proving with induction is difficult to automate, and is not supported by the proof scripts, inductively defined types should for example be avoided as much as possible. In the mechanised Isabelle verifications of [PN02], lists are often used where array-like types would me more suitable for automated verification. In our PVS verification of the garbage collection algorithm we make extensive use of predicate subtypes. This has turned out to be a good choice as it enables us to prevent many unwanted instantiations in PVS, using just a slightly modified proof script. More proof obligations of this algorithm could be proved automatically using this modified proof script.

Runtime The runtime of highly interactive proofs is usually shorter than those of automated proofs, but more human effort is required for the interactive proofs. The runtime of most of the PVS verifications we have experimented with is shorter than the runtime of the corresponding Isabelle verification. For most of the verifications this difference was marginal, but we have also seen an exception where the PVS verification is more than ten times faster.

Our conclusions are not completely in line with those of [GH98]. In [GH98], it is concluded that the automation offered by PVS and Isabelle is comparable. Our experience is that PVS offers more automation. The reason for this difference is most likely that the conclusions of [GH98] are mostly based on experiences with rewriting [Hui01].

Despite the fact that many non-trivial proof obligations can be proved fully automatically, we have also encountered proof obligations that are trivially valid but cannot be proved automatically using the theorem provers. Provers are powerful, but the power of human intelligence and creativity should not be underestimated.

Recommendations for further work

We have been working with the theorem prover in the role of user. We have tried to develop effective and generic proof scripts, using the tactics and rules available. We expect that there is not much more to gain from this perspective; we do not think it is useful to continue tuning the generic proof scripts.

Our advise for further work is to change to the role of developer, and try to improve the *provers*. The most prominent problem is how the theorem provers deal with quantifiers. Automated proofs with quantifiers are not trivial, but the instantiation heuristics used by PVS and Isabelle are rather primitive. We think it is worthwhile to investigate the possibilities of more intelligent instantiation heuristics.

The decision procedures of the provers are not perfect (this is especially true for Isabelle). Decision procedures are used to automatically decide which direction an automated proof (attempt) should head. We think it may be worthwhile to try to develop more effective decision procedures.

Bibliography

[AFP] Archive of Formal Proofs URL: `http://afp.sf.net/`.

[Con04] Contributing Members of the UPnP Forum. UPnP^{TM} Power Management 1.0, September 2004.

[FvG99] W. H. J. Feijen and A. J. M. van Gasteren. *On a method of multiprogramming*. Springer-Verlag, 1999.

[GH98] David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In *Theorem Proving in Higher Order Logics, TPHOLs '98*, volume 1479, pages 123–142. Springer-Verlag, 1998.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing. Springer-Verlag, 1995.

[Hee04a] Heerink, L. Deep Sleep Proxy. Philips, September 2004.

[Hee04b] Heerink, L. and Guidi, J. Deep Sleep Proxy Functionality. Philips, October 2004.

[Hes98] W. H. Hesselink. Invariants for the construction of a handshake register. *Information Processing Letters*, 68:173–177, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hui01] Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. Ipa dissertation series, 2001-03, University of Nijmegen, Holland, February 2001.

[Moo02] A. J. Mooij. Formal derivations of non-blocking multiprograms. Computer Science Report 02-13, Technische Universiteit Eindhoven, October 2002.

[MW03] Arjan J. Mooij and Wieger Wesselink. A formal analysis of a dynamic distributed spanning tree algorithm. Computer Science Report 03-16, Technische Universiteit Eindhoven, December 2003.

[MW05] Arjan J. Mooij and Wieger Wesselink. Incremental verification of Owicki/Gries proof outlines using PVS. In Kung-Kiu Lau and Richard Banach, editors, *Proceedings of Formal Engineering Methods: 7th International Conference, ICFEM 2005*, volume 3785 of *LNCS*, pages 390–404. Springer-Verlag, 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th Conference on Automated Deduction*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, 1992.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Pau99] L.C. Paulson. A Generic Tableau Prover and its Integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, March 1999.

[PN02] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

[PNG] Garbage Collection Theory from [PN02]. URL: `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/library/HOL/HoareParallel/Gar_Coll.html`.

[SORSC01] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[SUG] PVS Suggestion List. URL: `http://pvs.csl.sri.com/cgi-bin/pvs-suggest-list/`.

[UPn03] UPnP Forum. UPnP^{TM} Device Architecture 1.0, December 2003.

Appendix A

UPnP Power Management modelled in μCRL

A.1 Introduction

Universal Plug and Play (UPnP) technology [UPn03] defines an architecture for peer-to-peer network connectivity of devices. It is designed to bring easy to use, flexible, standards based connectivity to managed and ad-hoc or unmanaged networks. It is also designed to support zero-configuration, "invisible" networking and automatic discovery for a breadth of device categories from a wide range of vendors. In the UPnP forum, the technical committee is working on a specification of a UPnP power management standard [Con04] that allows devices to enter various power saving modes.

There are various proposals for proxy functionality for devices in sleep mode. We consider an attempt by Philips: the *Deep Sleep Proxy*, which is described in [Hee04a].

We are interested in various properties of UPnP power management and the *Deep Sleep Proxy*. To this end we have created models of various environments, using the modelling language μCRL [GP95]. μCRL is a process algebraic language and the μCRL toolset includes a simulator and an instantiator which generates the state-spaces of models (i.e. enumerate all possible states), which can then be analysed. In this report the models and the evaluation of the properties are discussed.

A.2 UPnP

The architecture document [UPn03] describes the possibility for (physical) devices and root devices to contain sub-devices and services. We limit our models to the following two logical entities:

Control point: A control point (or **CP**) functions in the role of client. A control point requests services from a controlled device, and can initiate searches for devices and services.

Controlled device: A controlled device (or simply **device**) functions in the role of server. A device offers and advertises a service. The device responds to search and service requests from the control points.

We abstract from physical devices; we do not specify how the logical entities are mapped to physical devices and we do not define how they are connected, what media connects them. Each device and control point is identified by a unique id, and it is assumed that they are all able to communicate with each other.

The following messaging services are of interest:

UDP Unicast/Multicast: UDP is an asynchronous, unreliable protocol. Messages are not acknowledged; they can get lost without being detected. Also, messages do not necessarily arrive in the same order they were sent. Both unicast and multicast messages are supported by UDP. A device can join multicast groups. Messages addressed to such a group can be received by the members of the group. In order to prevent an explosion of the state-space, a multicast message is either delivered to each entity but the sender, or gets completely lost and is not received by any entity. Note that without this restriction, there are $O(2^n)$ possible outcomes for every multicast message, with n being the number of possible receivers. UDP is modelled as a separate message queue for each entity, with each queue having a bounded size. Messages are read from the queue in a random order and can get lost. Sent messages are discarded if the queue of the receiver is full or if the receiver is offline.

TCP: TCP is a reliable protocol. We only use TCP unicast, which is used to request services and to send service results. It is modelled like RPC: the initiator makes a request, this either succeeds or fails (cannot connect or the receiver cannot accept the request). The receiver of a request either replies, or cannot reply for some reason, and the request remains unanswered. The initiator always knows whether the request is received, but there is no guarantee that it is ever processed or answered. For the sake of simplicity, the devices process requests in the order they were received and there can be at most one pending request per service, per control point. The reason for these simplifications is that we do not aspire modelling a complete tcp stack. The device does not 'block' while processing a request; it is able to do other things in the mean time.

We are especially interested in the discoverability of the devices and we limit our models accordingly. For UPnP (legacy) devices and control points we consider the following messages:

message	**from**	**to**	**using**
SSDP discovery messages			
`alive`	device	CPs	UDP Multicast
`byebye`	device	CPs	UDP Multicast
`msearch`	CP	devices	UDP Multicast
`msearch response`	device	CP	UDP Unicast
Control actions			
`service request`	CP	device	TCP
`service result`	device	CP	TCP

A device announces its presence with `alive` messages, and announces its departure with a `byebye` message. The `msearch` message is used by a control point to actively search for devices that offer a certain service. The `msearch response` message is a response to this: it is used by a device to report that it offers the service. A `service request` message is used to request a service, note that this is communicated synchronously. The `service result` message represents the result of the service, which is sent after the service has been delivered.

A.3 UPnP Power Management

In this section we discuss the specification of devices and control points that implement, or are aware of, power management.

We consider the following additional power modes for controlled devices, as mentioned in [Con04]: *Deep Sleep Online* and *Deep Sleep Offline*. In the *Deep Sleep Offline* mode and in the *Disconnected* mode there is no IP connectivity. In *Deep Sleep Online* mode, the functionality of a device is limited.

power mode	discoverable	controllable
Active	Yes	Yes
Deep Sleep Online	Yes	only power mode changes
Deep Sleep Offline	No	No
Disconnected	No	No

In addition to the messages listed in the previous section, we need additional messages to announce transitions to the new power modes. We introduce the following messages:

message	from	to	using
SSDP discovery messages			
`byebyehello`	device	CPs	UDP Multicast
`byebyebye`	device	CPs	UDP Multicast
`msearchsleep`	CP	devices	UDP Multicast
`msearchsleep result`	device	CP	UDP Unicast
Control actions			
`wake-up`	CP	device	TCP

A device that goes into *Deep Sleep Online* mode announces this by sending a `byebyehello` message. A device that goes into *Deep Sleep Offline* mode announces this by sending a `byebyebye` message. Legacy UPnP devices interpret the `byebyehello` messages and the `byebyebye` messages as `byebye` messages. Devices in *Deep Sleep Online* mode do not respond to `msearch` messages; they respond to `msearchsleep` messages instead [Hee04a]. The `msearchsleep result` message is used to respond to the control point. Devices in other states do not respond to `msearchsleep` messages. This way, from the point of view of a legacy control point, a device and its service simply do not exist (or are considered to be *Disconnected*) when they are in one of the sleep modes. This ensures compatibility.

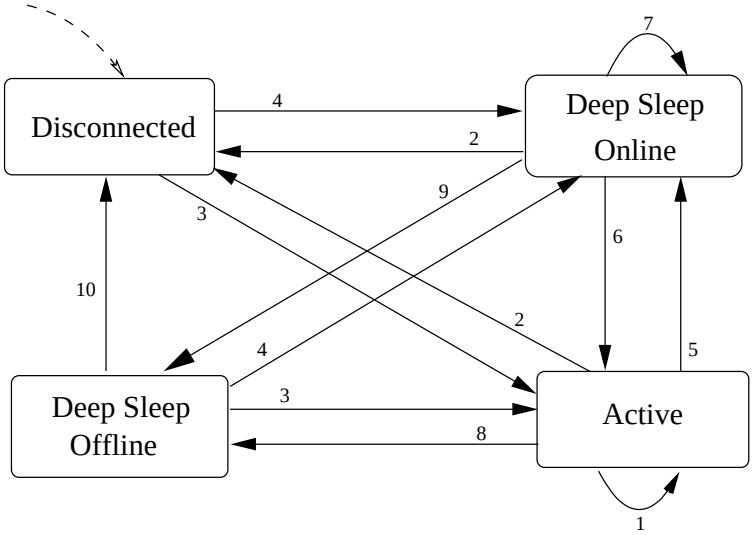

Figure A.1: Transition diagram of a device that implements power management

Figure A.1 shows the transition diagram of a power aware controlled device, and Table A.1 shows the corresponding triggers and responses. Several transitions have multiple triggers and multiple responses. Messages received in states where there is no corresponding transition in the transition diagram are discarded. *Internal* denotes an internal decision of the device to go to a certain state. Every device maintains a variable named `BootID`. The value of this variable is incremented each time a device is reconnected [Con04]. The last column shows which transitions

must increment the value of `BootID` before sending the response (see [Con04]). This device is able to go to sleep and wake up periodically, which is considered to be optional behaviour.

#	Trigger	Response	Inc BootID?
1	(Internal)	Multicast `alive`	No
	Receive relevant `msearch`	Unicast `msearch response`	No
	Receive relevant `service request`	Send `service result`	No
2	(Internal)	(no response)	No
	(Internal)	Multicast `byebye`	No
3	(Internal)	Multicast `alive`	Yes
4	(Internal)	Multicast `byebyehello`	Yes
5	(Internal)	Multicast `byebye`-hello	No
6	(Internal)	Multicast `alive`	Yes
	Receive `wake-up`	Multicast `alive`	Yes
7	Receive relevant `msearchsleep`	Unicast `msearchsleep result`	No
8	(Internal)	Multicast `byebyebye`	No
9	(Internal)	Multicast `byebyebye`	No
10	(Internal)	(nothing is sent)	No

Table A.1: Transitions of Figure A.1

A.4 Environment: One CP and one Device

In this section we discuss a model of an environment with one power aware device and one power aware control point. The device offers one service: `service1`, and the control point searches continuously for devices that offer this service and requests this service from available devices it has located. When the control point has located a device that offers the service, but the device is in *Deep Sleep Online* mode, it can attempt to wake-up the sleeping device. The device can decide (internal decision) to change states as described in Figure A.1. In Section A.4.1 a μCRL model of the environment is discussed. In Section A.4.2 the `BootID` variable is discussed.

A.4.1 μCRL model

Table A.2 shows the μCRL datatypes used and Table A.3 shows the (atomic) actions. Note that `s: devID` denotes a variable `s` with datatype `devID` and that the variable name is often omitted. The first column shows for each action whether it results from synchronised communication between processes.

Actions that are the result of communication have a sender action (prefix `s_`) and a receiver action (prefix `r_`) in the μCRL specification. This synchronised communication is shown in Figure A.2. The arrows show the direction (sender and receiver). The actions in the transitions are atomic.

The following processes are used:

comm1(qsUDP: IMQList, areOnline: devSet) This process models the UDP communication between devices and control points. The communication channel accepts UDP messages and delivers them to the receiver, unless they get lost. `qsUDP` is a list of message queues, one for each device and each control point. The process maintains a list of online devices (`areOnline`). Multicast messages, and the `disconnect` action, are used to maintain this list. Messages addressed to devices that are not online are discarded.

dev1(id: devID, state: devState, serviceQ: ISEQueue) This process models a device that implements all the states and transitions shown in Figure A.1. `id` is the unique identifier for this device, and `state` is the current state of the device. Received messages that do not

Datatype	Description	Possible values
`Bool`	Booleans	{`T`,`F`}
`Nat`	Natural numbers	0...
`service`	Services offered	{`service1`}
`messageType`	Message type	{`alive`, `byebye`, `byebyehello`, `byebyebye`, `msearch`, `msearchres`, `msearchsleep`, `msearchsleepres`}
`devID`	Unique ID for a device or CP	{`CP1`,`DEV1`}
`message`	UDP message	(`devID`, `messageType`, `service`)
`devState`	States a device can be in	{`disconnected`, `deepsleeponline`, `deepsleepoffline`, `active`}
`tIS`	Tuple of `devID` and `devState`	(`devID`, `devState`)
`tIMQ`	Tuple of `devID` and `mMulti`	(`devID`, `mMulti`)
`tISE`	Tuple of `devID` and `service`	(`devID`, `service`)
`mMulti`	Multiset of messages	Ordered list of messages with a maximum size. Same messages can appear more than once in the set. An ordered list is used so that there is a unique representation for every possible combination.
`IMQList`	List of `tIMQ`	[(`devID`, `mMulti`)]
`ISSet`	Set of `tIS`	Ordered list of `tIS`. Every device is at most once in the list.
`ISEQueue`	FIFO queue of `tISE`	The (`devID`, `service`) combination is unique in the queue (appears at most once).
`devSet`	Set of `devID`	Every device is at most once in the list. The list is ordered to get a unique representation.

Table A.2: μCRL abstract data types

correspond to a transition in the current state are read but discarded. A device can only be woken up when it is *Deep Sleep Online* mode. In any other state, a control point's attempt to wake up the device will fail. `serviceQ` is a queue of control points that are still waiting for a reply to a certain service. Each combination of control point and service appears at most once in the list. The device can only accept service requests when it is in state *Active*. Accepted service requests are put in the queue and requests are processed in FIFO order. When the device changes states, the `serviceQ` is cleared and pending service requests remain unanswered. The state of the device is initially *Disconnected* and the `serviceQ` is initially empty.

cp1(id: devID, serviceQ: ISSet) This process models a control point whose sole purpose is requesting service `service1` from devices that offers it. `id` is the unique identifier for this control point and `serviceQ` is a list of devices of which is known that they offer `service1`, together with their current state. Devices that are in state `disconnected` are not in the list. Messages that are not related to `service1` are received but discarded. Received multicast messages are used to maintain `serviceQ`. The control point can try to request the service from devices that are online and are offering the service. If there are no devices online that offer `service1`, then the control point can multicast a `msearch` message or a `msearchsleep` message to search for devices that do. The control point can attempt to wake-up devices that offer `service1`, but are in *Deep Sleep Online*.

comm?	Action	Description
No	`incBootID(devID)`	The device increases its `BootID`
Yes	`sendUC(s: devID, r: devID, messageType, service)`	Sender s sends an Unicast UDP message to r.
Yes	`sendMC(devID, messageType, service)`	The device or CP sends a multicast message.
Yes	`read(r: devID, s: devID, messageType, service)`	r receives a message which is sent by s.
Yes	`wakeup(devID,Bool)`	The device is asked to wake up. Bool indicates success (T) or failure (F).
No	`serviceTimeout(s: devID, r: devID, service)`	Device s was not able to fulfill the service request of r. Reply is never sent. This action models the loss of the request.
Yes	`disconnected(devID)`	Internal decision of the device to go to disconnected.
Yes	`serviceReq(s: devID, r: devID, service, Bool)`	CP s requests a service of r. Bool indicates whether r was able to accept the request.
Yes	`serviceRep(s: devID, r: devID, service)`	Device s sends a reply to the request of a service by CP r.

Table A.3: μCRL actions

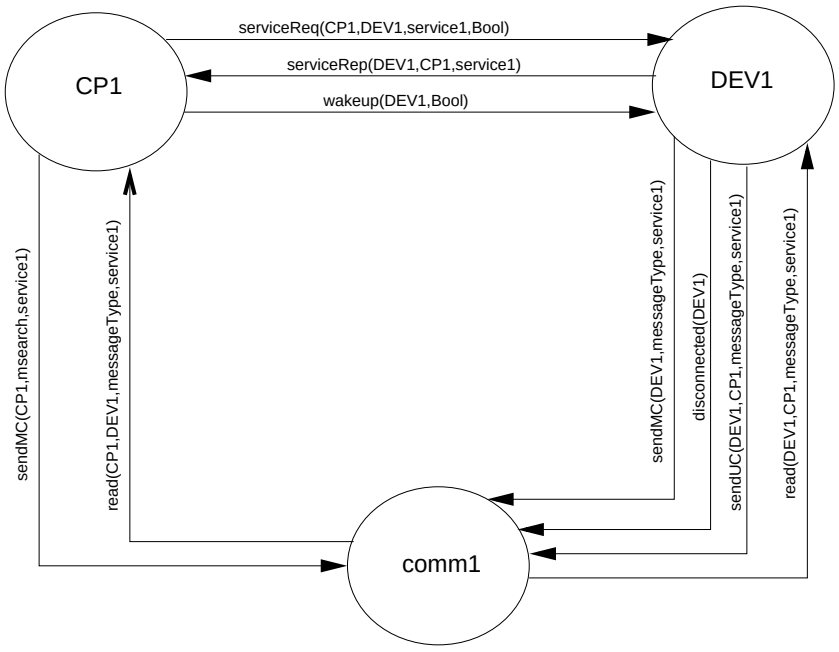

Figure A.2: Communication between processes

State-spaces

The size of the state-spaces that are generated depends heavily on the maximum size of the queues used for the UDP communication. We have experimented with different queue sizes to determine

up to what size we can generate and reduce the state-spaces. For the instantiation and reduction of the state-space we made use of a server running Linux with a Intel Xeon 2.8GHZ CPU (5500 bogomips). A maximum of 4GB of RAM is available on this server. Table A.4 shows the number of states and transitions before and after reduction (branching bisimulation). Table A.5 gives an impression of the amount of the CPU time, and maximum memory usage, for the instantiation and reduction. More combinations are discussed in Appendix B. We were not able to produce and reduce the state-space of models with a queue size above 7.

	Original		**Reduced**	
Queue size	*States*	*Transitions*	*States*	*Transitions*
1	547	3K	383	2K
2	5K	32K	3K	22K
3	24K	201K	17K	127K
4	91K	889K	67K	531K
5	284K	3.1M	212K	1.8M
6	777K	9.4M	574K	5.0M
7	1.9M	24.9M	1.4M	12.5M

Table A.4: Sizes of the state-spaces

	Instantiation		**Reduction**	
Model	*CPU*	*Mem*	*CPU*	*Mem (KB)*
1	6s	6MB	0s	8MB
2	14s	7MB	1s	9MB
3	1m11s	12MB	5s	17MB
4	5m8s	26MB	25s	47MB
5	20m1s	68MB	2m24s	136MB
6	1h5m	180MB	8m54s	354MB
7	3h1m	427MB	26m37s	878MB

Table A.5: Resources used for generation and reduction

A.4.2 BootID

Every device maintains a variable `BootID`. This variable is incremented each time a device reconnects to the network [Con04]. Different interpretations of the desired property of the `BootID` variable are possible, depending on how the states of the devices are classified. In this section we examine the different interpretations.

Different interpretations

1. The state of a device is classified as either *online* or *offline*. Transitions between states in the same category do not have any influence on `BootID`. After a transition from a state in the category *online* to a state in the category *offline*, there must be an `incBootID` action before the next transition to a state in *online*. It then depends on which states belong to *online* and which states belong to *offline*:

   (a) We classify *online* and *offline* as:
   online = {*Active*, *Deep Sleep Online*}
   offline = {*Deep Sleep Offline*, *Disconnected*}

The actions that indicate a transition from *online* to *offline* are (transitions 2, 8 and 9 in Figure A.1):

- `sendMC(DEV1,byebye,service1)`
- `sendMC(DEV1,byebyebye,service1)`
- `disconnected(DEV1)`

And the actions that indicate a transition from *offline* to *online* are (transitions 3 and 4 in Figure A.1):

- `sendMC(DEV1,alive,service1)`
- `sendMC(DEV1,byebyehello,service1)`

(b) We classify *online* and *offline* as:
online = {*Active*}
offline = {*Deep Sleep Online*, *Deep Sleep Offline*, *Disconnected*}
The actions that indicate a transition from *offline* to *online* are (transition 3 and 6 in Figure A.1):

- `sendMC(DEV1,alive,service1)`

The actions that indicate a transition from *online* to *offline* are (transitions 2, 5 and 8 in Figure A.1):

- `sendMC(DEV1,byebye,service1)`
- `sendMC(DEV1,byebyebye,service1)`
- `sendMC(DEV1,byebyehello,service1)`
- `disconnected(DEV1)`

2. The state of a device belongs to one of the following categories:
online = {*Active*}
limited-online = {*Deep Sleep Online*}
offline = {*Disconnected*, *Deep Sleep Offline*}
See Figure A.3 and Table A.6. There must be an `incBootID` action before each up-arrow (4,5,6). More specifically, after an action in {`sendMC(DEV1,`t`,service1)`, `disconnected(DEV1)`} with $t \in$ {`byebye`, `byebyebye`, `byebyehello`}, there must be an `incBootID(DEV1)` before the next action in {`sendMC(DEV1,`t'`,service1)`} with $t' \in$ {`alive`, `byebyehello`}.

#	**Action**
1	`sendMC(DEV1,byebyehello,service1)`
2	`sendMC(DEV1,byebye,service1)`
	`disconnected(DEV1)`
3	`sendMC(DEV1,byebyebye,service1)`
	`disconnected(DEV1)`
4	`sendMC(DEV1,byebyehello,service1)`
5	`sendMC(DEV1,alive,service1)`
6	`sendMC(DEV1,alive,service1)`

Table A.6: Transitions of Figure A.3

Options examined

CADP's evaluator has been used to check the different properties.

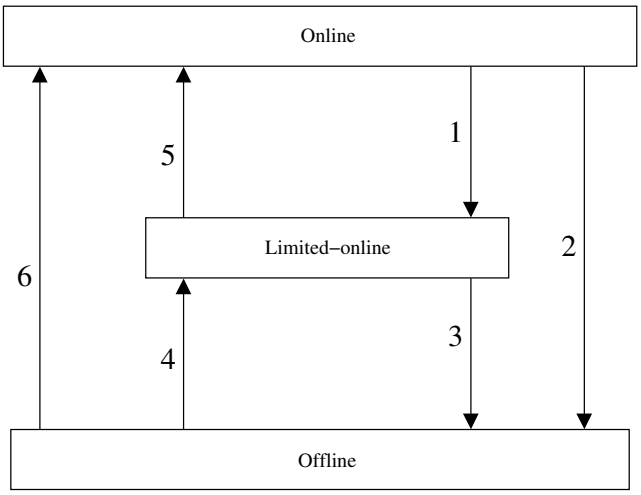

Figure A.3: BootID option 2

Option 1a This option can be described as: for every possible path it is not allowed that there is an action in {`sendMC(DEV1,byebye,service1)`, `sendMC(DEV1, byebyebye, service1)`, `disconnected(DEV1)`} on this path after which there is no `incBootID(DEV1)` before the next multicast of this device. This gives the following formula:

```
[true* . (      "sendMC(DEV1,byebye,service1)"
            |  "sendMC(DEV1,byebyebye,service1)"
            |  "disconnected(DEV1)"
         ) . (not("incBootID(DEV1)"))* . 'sendMC(DEV1,.*)' ] false
```

The `BootID` variable is used by both legacy and power aware devices. Since legacy CPs do not distinguish the deep sleep modes from the *Disconnected* mode, option 1a is too weak for legacy CPs. For example, when the device goes from *Active* to *Deep Sleep Online* (from the point of view of the legacy control point the device is disconnected) and then goes back to *Active*, the legacy control point expects an increment of `BootID` in between the actions, but option 1a does not enforce this. In other words, the sequence

`sendMC(DEV1,byebyehello,service1).sendMC(DEV1,alive,service1)`

is not forbidden by this option, but this path should not be allowed.

Option 1b This option can be described as: for every possible path it is not allowed that there is an action in {`sendMC(DEV1, byebye, service1)`, `sendMC(DEV1,byebyebye,service1)`, `disconnected(DEV1)`} on this path after which there is no `incBootID(DEV1)` before the next `sendMC(DEV1, alive, service1)`. This gives the following formula:

```
[true* . (      'sendMC(DEV1,byebye,service1)'
            |  'sendMC(DEV1,byebyebye,service1)'
            |  'sendMC(DEV1,byebyehello,service1)'
            |  "disconnected(DEV1)"
         ) . (not("incBootID(DEV1)"))* . 'sendMC(DEV1,alive,service1)' ] false
```

This option would work for the legacy CPs as it treats all states but *Active* as *offline*. However, this option is too weak for the power aware CPs. For example, a device in *Deep Sleep Online* changes states to *Disconnected*, and later it changes back to *Deep Sleep Online*. This gives the sequence:

`sendMC(DEV1,byebye,service1).sendMC(DEV1,byebyehello,service1)`

There is no `incBootID(DEV1)` action in between these two actions. This is not forbidden

by option 1b, because both states belong to *offline*. But this should be forbidden, since the device has re-entered the network. See also [Con04, Section 1.2].

Option 2 This option can be described as: for every possible path it is not allowed that there is an action in {`sendMC(DEV1, byebye, service1)`, `sendMC(DEV1, byebyebye, service1)`, `sendMC(DEV1, byebyehello, service1)`, `disconnected(DEV1)`} on this path after which there is no `incBootID(DEV1)` before the next action in {`sendMC(DEV1, alive, service1)`, `sendMC(DEV1, byebyehello, service1)`}. This gives the following formula:

```
[true* . (      'sendMC(DEV1,byebye,service1)'
           |    'sendMC(DEV1,byebyebye,service1)'
           |    'sendMC(DEV1,byebyehello,service1)'
           |    "disconnected(DEV1)"
         )
. (not("incBootID(DEV1)"))* . ( 'sendMC(DEV1,alive,service1)'
                              | 'sendMC(DEV1,byebyehello,service1)') ] false
```

This option seems to be consistent with the second table of [Con04, Section 1.2]. We examine the offending sequences of the other two options:

1. `sendMC(DEV1,byebyehello,service1).sendMC(DEV1,alive,service1)`
   This sequence is forbidden if option two is used, since there is no `incBootID(DEV1)` in between: between transition 1 and the next 5 in Figure A.3 there must be at least one `incBootID(DEV1)`.
2. `sendMC(DEV1,byebye,service1).sendMC(DEV1,byebyehello,service1)`
   This sequence is forbidden if option two is used, since there is no `incBootID(DEV1)` in between: between transition 3 and the next transition 4 in Figure A.3 there must be at least one `incBootID(DEV1)`.

All paths that are allowed by option 2 are also allowed by option 1a and option 1b. Option 2 is stricter than option 1a and 1b.

We believe option two is the correct interpretation. We have checked the formulas of the different options for this model, they are all satisfied.

A.5 Including Deep Sleep Proxy functionality

An attempt at proxy functionality is given by Philips in [Hee04a]. The *Deep Sleep Proxy* they propose basically has two interesting functions: First, the proxy keeps track of all the devices on the network, together with the services they offer and the (power) states they are in. A CP (both power aware and legacy) can send search requests to the proxy, requesting a list of devices or services available, and the state they are in. Second, the proxy can be used to wake-up devices in *Deep Sleep Offline* mode. This is only possible if the specific proxy has some mechanism to wake-up the specific device. The method used by the proxy for the actual wake-up of the device is not specified and could be anything. From now on we refer to this wake-up as a "link-layer wake-up".

We introduce the following UPnP messages:

message	**from**	**to**	**using**
Control actions			
`search request`	CP	Proxy	TCP
`search response`	Proxy	CP	TCP
`link-layer wake-up request`	CP	Proxy	TCP
`link-layer wake-up`	Proxy	Device	Unspecified

Note that there is no response to a `link-layer wake-up request`. This is because [Hee04b] states that some link-layer wake-up methods do not support acknowledgements. The CP should

wait for discovery messages. We limit our search requests to requests for services, i.e. asking the question: "Which devices offer service X and what is their state?" The Proxy has two power modes: `on` and `off`. See Figure A.4 and Table A.7 for the behaviour of the proxy.

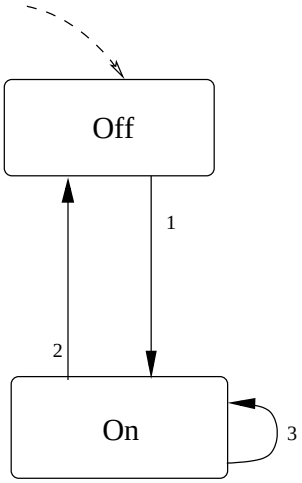

Figure A.4: Transition diagram of a deep sleep proxy

#	**Trigger**	**Response**
1	(Internal)	Multicast `alive`
2	(Internal)	Multicast `byebye`
3	(Internal)	Multicast `alive`
	Receive `msearch` for proxy service	Unicast `msearch response`
	Receive `link-layer wake-up request`	Wakeup device
	Receive `search request`	Send `search response`
	Receive `alive`	Device is in *Active*, update internal table
	Receive `byebye`	Device is in *Disconnected*, update internal table
	Receive `byebyehello`	Device is in *Deep Sleep Online*, update internal table
	Receive `byebyebye`	Device is in *Deep Sleep Offline*, update internal table

Table A.7: Transitions of Figure A.4

Figure A.5 and Table A.8 show the behaviour of a device that supports link-layer wake-up. The changes are in **bold** in both the figure and the table.

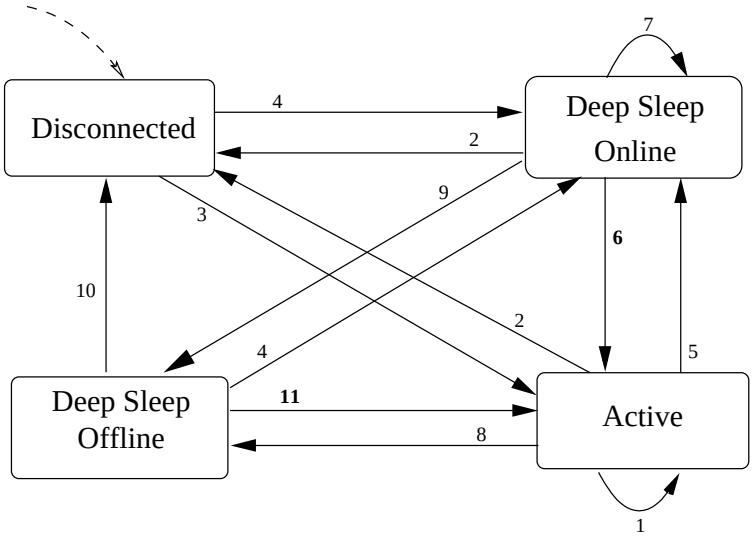

Figure A.5: Transition diagram of a device that supports link-layer wake-up

#	Trigger	Response	Inc BootID?
1	(Internal)	Multicast `alive`	No
	Receive relevant `msearch`	Unicast `msearch response`	No
	Receive relevant `service request`	Send `service result`	No
2	(Internal)	(no response)	No
	(Internal)	Multicast `byebye`	No
3	(Internal)	Multicast `alive`	Yes
4	(Internal)	Multicast `byebyehello`	Yes
5	(Internal)	Multicast `byebyehello`	No
6	(Internal)	Multicast `alive`	Yes
	Receive `wake-up`	Multicast `alive`	Yes
	Receive link-layer wake-up	**Multicast alive**	**Yes**
7	Receive relevant `msearchsleep`	Unicast `msearchsleep result`	No
8	(Internal)	Multicast `byebyebye`	No
9	(Internal)	Multicast `byebyebye`	No
10	(Internal)	(nothing sent)	No
11	(Internal)	Multicast `alive`	Yes
	Receive link-layer wake-up	**Multicast alive**	**Yes**

Table A.8: Transitions of Figure A.5

The following μCRL datatypes are modified:

Datatype	Description	Possible values
`service`	Various services offered	{`service1`, `proxyserv`}
`devID`	Unique ID for every device, CP or proxy	{`CP1`,`CP2`,`DEV1`,`DEV2`,`PROXY1`,`PROXY2`}

Table A.9 shows the μCRL actions that have been added in order to support and implement Deep Sleep Proxy functionality.

The dev1 process has been modified slightly:

dev1 (id: devID, state: devState, serviceQ: ISEQueue, offers: service)

comm?	Action	Description
No	searchTimeout(s: devID, r: devID, service)	Proxy s could not send a reply to search request of r in time. Reply is never sent.
No	llwakeupTimeout(s: devID, r: devID)	Proxy s was not able to wake-up (link-layer) device r in time.
Yes	llwakeupReq(s: devID, r: devID, d: devID, Bool)	Device s sends a request to proxy r to wake-up (link-layer) device d. Bool indicates success or failure of the request.
Yes	connected(s: devID)	CP s is connected.
Yes	searchReq(s: devID, r: devID, service, Bool)	Device s requests the list of devices that offer the service and their states from proxy r.
Yes	searchRep(s: devID, r: devID, service, ISSet)	Proxy s sends the list of devices (ISSet) that offer service and their states to CP r.

Table A.9: μCRL actions for deep sleep proxy

`Offers` is the service offered by this device. It has been introduced to minimise the effort required to add devices that offer different services. Link-layer wake-up has been added, as shown in Table A.8.

The following processes have been added to the μCRL specification:

comm2 (qsUDP: IMQList, areOnline: devSet)
The behaviour of `comm2` is almost identical to that of `comm1`. The only difference is that we lifted the restriction that multicast messages either reach all subscribed entities, or none. Multicast messages are delivered to a subset of `areOnline`.

cp2 (id: devID, serviceQ: ISSet, proxyQ: devSet, online: Bool)
The behaviour of this process is similar to that of `cp1`. There are two important additions: First, this control point is not always online, it is either online or offline. The process informs the communication channel when it changes states. When a control point goes offline the administration of available devices is cleared: `serviceQ` and `proxyQ` are cleared. Second, this control point is able to use proxies to search for devices and their power state, and to wake-up sleeping devices. `proxyQ` is a list of all known online proxies. If both the `serviceQ` and the `proxyQ` are empty, then the CP can send an msearch multicast message to search for proxies. If the `serviceQ` is empty, but there are proxies online, then the CP can send search requests to one of the online proxies. When the control point receives a reply to a search request from a proxy, it assumes that the newly required information is accurate and uses this information for it's internal administration (`serviceQ`).

When there are proxies online, but all devices known to offer `service1` are in *Deep Sleep Offline*, then one of the online proxies can be used to wake-up the device (link-layer wake-up request).

proxy1 (id: devID, serviceQ: ISSet, online: Bool, searchRequests: devSet, wakeupRequests: devSet)
This process models a *Deep Sleep Proxy*. `id` is a unique identifier for this proxy. `serviceQ` is a list of devices that offer `service1`, and their state. A proxy can be either online or offline. `online` is the current state. When the proxy goes offline, the internal tables `searchRequests` and `wakeupRequests` are cleared. The proxy offers the proxy service, named `proxyser`. An alive message is sent when going online, and a byebye message is sent

when going offline. Also, like any other service, alive message are sent periodically while being online. `searchRequests` is a list of devices waiting for a reply to a search request. The search requests are handled in random order, and every control point has at most one pending request for `service1`. Officially the proxy should keep a list of all services and the devices that offer them together with their state. But, to keep it manageable, we only keep track of `service1` and only support search requests for this service. `wakeupRequests` is a list of devices to wake-up. Wake-up requests are handled in a random order. See Figure A.4 for the behaviour of this process.

A.5.1 Model: Including a proxy

In this section we discuss a model that includes a *Deep Sleep Proxy*. Figure A.6 shows processes and the communication between the processes. The cp2 process is used for CP1 because it supports proxies. Table A.10 lists the transitions of Figure A.6. Transition 11 has 2 receivers, this means that three processes have to synchronise. Three-way synchronised communication is not natively supported by μCRL, but it can be simulated: Imagine we have three parallel processes: P, Q and R. P sends something that is to be received by Q and R at the same time; it has to be one atomic action. P uses the action *send* and Q and R use the action *recv*. As usual, *send* and *recv* communicate to, say *c*. We then use something like:

act $send, recv, c, d$
proc $P = send.P$
proc $Q = recv.Q$
proc $R = recv.d.R$
comm $send \mid recv = c$
init $\partial_{\{send,recv\}}(\rho_{\{c \to send\}}(\partial_{\{send,recv\}}(P \parallel Q)) \parallel R)$

Which does exactly what we need. Note that this initial process is equivalent to the process Z:

proc $Z = c.d.Z$

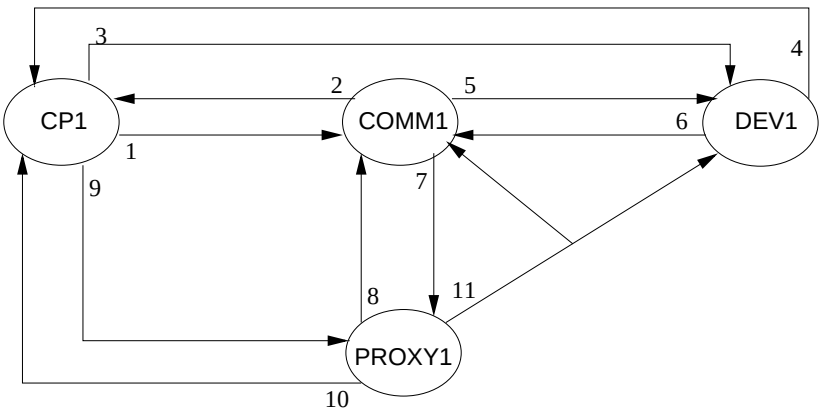

Figure A.6: Communication with proxy

State-spaces

We have tried different combinations of processes and queue sizes, see Appendix B for details. State-spaces can only be generated for simple environments, with few processes and small queue

#	Action
1	sendMC(CP1,msearch,*service*)
	connected(CP1)
	disconnected(CP1)
2	read(CP1,*devID*,*messageType*,service1)
3	serviceReq(CP1,DEV1,*Bool*)
	wakeup(DEV1,*Bool*)
4	serviceRep(CP1,DEV1,service1,*Bool*)
5	read(DEV1,*devID*,*messageType*,service1)
6	sendMC(DEV1,*messageType*,service1)
	sendUC(DEV1,CP1,*messageType*,service1)
	disconnected(DEV1)
7	read(PROXY1,*devID*,*messageType*,*service*)
8	sendMC(PROXY1,alive,proxyser)
	sendUC(PROXY1,CP1,msearchres,proxyser)
9	searchReq(CP1,PROXY1,service1)
	llwakeupReq(CP1,PROXY1,DEV1,*Bool*)
10	searchRep(PROXY1,CP1,service1,*ISSet*)
11	llwakeup(PROXY1,DEV1,*Bool*)

Table A.10: Transitions of Figure A.6

sizes. We are, for instance, not able to generate a state-space for models with more than one device in combination with a proxy. Even a model with one device and one proxy can only be generated if a queue size of one is used, which is not very representative for real network environments.

A.5.2 Evaluation of the Deep Sleep Proxy

In this section we examine the *Deep Sleep Proxy* more closely. We are especially interested in finding out in which situations the proxy has the correct information regarding the state of devices and – even more importantly – what the consequences are of having the wrong information; what it takes for the proxy to reflect the right state of a device again.

To this end, we use a model with one control point (cp2), one device (dev1), one Deep Sleep Proxy (proxy1) and a communication channel where messages can get lost arbitrarily (comm2). UDP messages sent arrive at a subset of the available devices, including the empty set, which reflects the situation where a message is completely lost. We use a queue size of 1 and remove the incBootID transitions.

We divide the states of the state-space of the model into equivalence classes where two states belong to the same class if, and only if, the state of the device is equal for both states, the state the proxy thinks the device is in is equal for both states, and the state (online/offline) of the proxy is the same for both states. It's trivial that every state belongs to exactly one class. In order to be able to identify the class for each state, we add a self loop (transition) to every state with information about the state.

The first thing we'd like to know is how many states belong to the different equivalence classes. To determine this we translate the state-space to a plain text representation, filter out the transitions that supply the state information and then use basic unix utilities (*grep* and *wc*) to count the number of occurrences for the different combinations. 3373 states out of the total of 451282 states belong to one of the equivalence classes where the proxy if offline. We are especially interested in states where the proxy is online. The following table shows the number of states of the different classes where the proxy is online.

proxy thinks	Actual state of device				Total
	Active	*DS Online*	*DS Offline*	*Disconnected*	
Active	54327	32717	5680	18214	110938
DS Online	54893	32893	5682	18556	112024
DS Offline	55490	33307	5778	18656	113231
Disconnected	55258	32068	5678	18712	111716
Total	219968	130985	22818	74138	447909

There are 111710 states where the proxy has the correct information regarding the device. That is 24.9% out of the 447909 states where the proxy is online. This obviously does not imply that the proxy has the correct information 24.9% of the time as we did not take the transitions into account. In each state each transition leaving that state has an equal probability of being selected. This obviously does not reflect the real situation.

From now on we'll call a state a "wrong state" if the state the proxy thinks the device is in does not equal the actual state of the device, and a "right" state otherwise. When in a wrong state, what does it take to get to a right state?

We replace the state information labels in the state-space with the labels "OK" and "NOK" for right (resp. wrong) states. The following formula is used to verify that it is always possible to go from a wrong state to a right state with a maximum of two steps:

```
[true* . "NOK"] ( (<true . "OK"> true) or (<true . true . "OK"> true))
```

This formula is satisfied – which may surprise the reader – because a device can always change to any power state within a few steps. So, it gets to a right state because the state of the device changes and not because the proxy is informed about the actual state of the device. If we only consider paths where the device does not change its state, then there are wrong states where there is no possibility of ever getting to a right state. The most obvious situation where this occurs is when the following happens:

- The device is offline, the proxy is offline, the CP is offline.
- The device comes online.
- The device goes into *Deep Sleep Offline*.
- The proxy comes online.
- The CP comes online.

After this scenario the device is in *Deep Sleep Offline* mode and neither the control point, nor the proxy knows about the device. As long as the device does not change states it wont be discoverable.

We are interested in finding, for each class, a number `n` for which from each wrong state in the class a right state can be reached, with a path without state changes of the device, with a length of at most `n`. Obviously there are classes for which there is no such `n`. We ignore the classes where the proxy is offline. The following table shows this `n` for every class.

proxy thinks	Actual state of device			
	Active	*DS Online*	*DS Offline*	*Disconnected*
Active	0	∞	∞	3
DS Online	6	0	∞	3
DS Offline	6	∞	0	3
Disconnected	6	∞	∞	0

The only reason the proxy gets to a right state when the device is disconnected is because the proxy can go offline and get back online again. When the proxy gets back it starts in a state where every device is considered to be disconnected. When the device is active the proxy discovers this when the device sends an `alive` multicast. The reason it still takes 6 steps to get to a right state, is that both the device and the proxy can be in a state where they are forced to do something else first before sending or receiving the message. Consider for example the following scenario:

– The proxy has just read a msearch request of the CP.
– The device has just read a msearch request of the CP.
– The CP sends another msearch request to the device and proxy (both arrive in the corresponding message queue).

Consider the state we are now in. The proxy has to do (at least) the following actions (in this order) to end up in a right state:

1. Reply to the msearch request of the CP.
2. Read the msearch request of CP.
3. Reply to the msearch request of CP.
4. Read the alive message of device.

The first action is a consequence of the way the proxy is modelled. After receiving a msearch, a reply is sent first, before reading the next message. This is not dictated by the standard so this step could be eliminated. The device has to do at least the following actions to get an alive message to the proxy:

1. Reply to msearch request of the CP
2. Send alive message

Again, the first action is a consequence of the way the replies to msearches are modelled. The reason there are states where the proxy cannot discover the device in *Deep Sleep Online* mode is that a device does not periodically send alive messages in *Deep Sleep Online* mode and the proxy does not actively search for devices in *Deep Sleep Online* mode.

We change the model of the proxy and add the possibility to actively search for devices in *Active* and *Deep Sleep Online* mode (sending msearch and msearchsleep requests). Also, we add the following behaviour to the proxy: if the proxy thinks the device is in *Active* mode, but does not receive an alive message in time (we don't specify the actual time), it internally changes the state of the device to *Disconnected*. After the changes we get the following results:

	Actual state of device			
proxy thinks	*Active*	*DS Online*	*DS Offline*	*Disconnected*
Active	0	10	∞	2
DS Online	6	0	∞	3
DS Offline	6	10	0	3
Disconnected	6	10	∞	0

After the changes the device is also discoverable by the proxy when it's in *Deep Sleep Online* mode. The longest paths, which require 10 actions, are similar to the ones mentioned for *Active* mode. The difference is that an extra request is required, and when the message queue of the proxy is full, an extra read and reply to the control point is required before the msearchsleep request can be received.

A.6 Conclusions and further work

We chose μCRL as modelling language because of its simplicity. The language turned out to be powerful; it is expressive enough for our models. However, each abstract datatype has to created from scratch by the user, even the representation of naturals and booleans. We have used various datatypes that represent lists and sets. It is likely it could have saved a lot of work if such common types were available standardly. Build-in types and operations for lists and sets would most likely also reduce the time needed to generate the state-space, because build-in types, and build-in operations on these types, are likely to be more efficient than those we created, using the equation rewrite system.

Model checking can be very useful. It is often more desirable to abstract from a real system, and use a model to reason about properties of the system. Properties of the system can be verified completely automatically, provided the model is accurate. But, as we have seen in the report, it also has its limits. Most notably is the well-known state-space explosion problem. In general a state-space is generated and properties are checked on this state-space. But, for many models the state-space cannot be generated because the state-space is too large, or infinite. Unfortunately, this problem often occurs with models of communication standards.

The desired properties of `BootID` were not clear, and were multi-interpretable. We have checked different interpretations and we have shown what we believe is the correct interpretation, and why we believe others to be incorrect. This helped in understanding the `BootID`.

We have evaluated the Deep Sleep Proxy and have highlighted some weaknesses of the proposal. Most notably, there are situations where even the proxy can't discover a device that is in *Deep Sleep Offline* mode.

Further work could be in the direction of quantitative analysis of environments. This way questions can be answered about the probability certain (undesired) events occur or about the length of time it takes before a proxy gets into a correct state. However, more statistical information about the used UPnP environments is required.

Appendix B

State-spaces of UPnP Power Management models

Table B.1 shows the number of states and transitions of the state-spaces of the different models. Queuesize is the maximum size of the message queues for the devices. Table B.2 shows how much CPU time and memory is used for the generation and reduction of the state-space. The server "poema.win.tue.nl" has been used for the generation and reduction of the state-spaces. This server has a Intel Xeon CPU 2.8ghz cpu (5500 bogomips) and has 4GB of memory.

Model	**Original**		**Reduced (strong bis)**	
	States	*Transitions*	*States*	*Transitions*
cp1, dev1, comm1, qs 1	547	3069	383	2077
cp1, dev1, comm1, qs 2	4541	32453	3301	21551
cp1, dev1, comm1, qs 3	23562	200592	17396	126940
cp1, dev1, comm1, qs 4	90645	888612	67405	531176
cp1, dev1, comm1, qs 5	284183	3139944	212111	1770008
cp1, dev1, comm1, qs 6	767585	9415285	574133	5009569
cp1, dev1, comm1, qs 7	1850484	24913120	1385904	12534364
cp1, dev1, comm2, qs 3	23562	200592	17396	126940
cp1, dev1, comm2, qs 4	90645	888612	67405	531176
cp1, dev1, comm2, qs 5	284183	3139944	212111	1770008
cp2, dev1, comm1, qs 1	673	4250	673	4225
cp2, dev1, comm1, qs 2	6824	54425	6824	54425
cp2, dev1, comm1, qs 3	43374	406184	43374	371448
cp2, dev1, comm1, qs 4	198358	2113831	198358	1822403
cp2, dev1, comm1, qs 5	720560	8578214	720560	6968944
2x cp2, dev1, comm1, qs 1	112944	1104463	47824	452896
cp2, dev1, comm2, qs 1	925	5860	925	5860
cp2, dev1, comm2, qs 2	8032	63411	8032	61009
cp2, dev1, comm2, qs 3	47386	440029	47386	401152
cp2, dev1, comm2, qs 4	208928	2213351	208928	1903188
cp2, dev1, comm2, qs 5	744348	8825293	744348	7154812
cp1, 2x dev1, comm1, qs 1	96471	803688	48343	397988
cp1, 2x dev1, comm1, qs 2	2923505	29607918	1555727	14818004
cp1, 2x dev1, comm2, qs 1	96847	816240	48443	401144
cp2, 2x dev1, comm1, qs 1	120879	1107229	118327	1087975
cp2, 2x dev1, comm2, qs 1	156956	1441222	154404	1421968
cp2, dev1, proxy1, comm1, qs 1	425342	4004498	425342	4004498
cp2, dev1, proxy1, comm2, qs 1	581806	5528779	573046	5447236

Table B.1: Size state-spaces

Model	Instantiation		Reduction	
	CPU (sec)	*Mem (KB)*	*CPU (sec)*	*Mem (KB)*
cp1, dev1, comm1, qs 1	5.24	6520	0.09	7848
cp1, dev1, comm1, qs 2	14.22	7380	0.57	9196
cp1, dev1, comm1, qs 3	71.10	11996	5.48	16556
cp1, dev1, comm1, qs 4	308.51	26776	24.95	47312
cp1, dev1, comm1, qs 5	1200.94	68720	144.49	136428
cp1, dev1, comm1, qs 6	3949.84	180360	476.81	353512
cp1, dev1, comm1, qs 7	10914.61	427776	1597.98	878340
cp1, dev1, comm2, qs 3	76.50	11940	5.35	16556
cp1, dev1, comm2, qs 4	324.49	26796	33.85	46821
cp1, dev1, comm2, qs 5	1240.43	68676	142.03	132512
cp2, dev1, comm1, qs 1	6.01	6576	0.11	6268
cp2, dev1, comm1, qs 2	21.73	8036	1.08	9408
cp2, dev1, comm1, qs 3	146.01	16220	14.42	30256
cp2, dev1, comm1, qs 4	818.60	54936	92.89	119068
cp2, dev1, comm1, qs 5	3570.85	180752	616.54	393868
2x cp2, dev1, comm1, qs 1	370.9	30060	34.75	48432
cp2, dev1, comm2, qs 1	6.03	6532	0.14	6256
cp2, dev1, comm2, qs 2	24.47	8112	1.28	11328
cp2, dev1, comm2, qs 3	158.24	16420	15.79	32948
cp2, dev1, comm2, qs 4	863.76	57316	113.40	122976
cp2, dev1, comm2, qs 5	3750.30	182780	618.42	401892
cp1, 2x dev1, comm1, qs 1	258.51	27128	25.57	41000
cp1, 2x dev1, comm1, qs 2	11530.97	698196	1828.88	1220556
cp1, 2x dev1, comm2, qs 1	278.65	27080	25.45	39180
cp2, 2x dev1, comm1, qs 1	348.26	32804	41.43	72640
cp2, 2x dev1, comm2, qs 1	461.53	38900	59.65	86820
cp2, dev1, proxy1, comm1, qs 1	1596.58	105336	257.86	251360
cp2, dev1, proxy1, comm2, qs 1	2229.77	123204	401.80	336036

Table B.2: Generation of state-spaces