Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL

Leonor Prensa Nieto

Institut für Informatik
Technische Universität München

Institut für Informatik
der Technischen Universität München
Lehrstuhl für Software & Systems Engineering

Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL

Leonor Prensa Nieto

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 31. Oktober 2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 31. Januar 2002 angenommen.

Abstract

This thesis presents the first formalization of the Owicki-Gries method and its compositional version, the rely-guarantee method, in a theorem prover. These methods are widely used for correctness proofs of parallel imperative programs with shared variables. We define syntax, semantics and proof rules in Isabelle/HOL, which is the instantiation of higher-order logic in the theorem prover Isabelle. The proof rules also provide for programs parameterized in the number of parallel components. Their correctness w.r.t. the semantics is proven mechanically and the completeness proofs for both methods are extended to the new case of parameterized programs. For the automatic generation of verification conditions we define a tactic based on the proof rules. Using this tactic we verify several non-trivial examples for parameterized and non-parameterized programs.

Zusammenfassung

In dieser Arbeit wird die Owicki-Gries Methode, und ihre kompositionelle Version, die Rely-Guarantee Methode, zur Verifikation paralleler imperativer Programme mit gemeinsamen Variablen zum ersten Mal in einem Theorembeweiser formalisiert. Syntax, Semantik und Beweisregeln werden in höherstufiger Logik definiert und die Korrektheit des Beweissystems bezüglich der Semantik wird bewiesen. Zahlreiche Beispiele, darunter parametrisierte parallele Programme, werden mit Hilfe einer Taktik für die systematische Generierung der Verifikations-Bedingungen verifiziert. Außerdem wird die Vollständigkeit der formalisierten Systeme für den Fall parametrisierter paralleler Programme bewiesen.

Acknowledgements

First of all I wish to thank Tobias Nipkow and Javier Esparza for giving me the opportunity to stay in Germany and work on my Ph.D. under their supervision. During four years they have continuously given direction to my work, motivated me and given me advice whenever I had problems.

Special thanks also to David von Oheimb, Markus Wenzel and Cornelia Pusch who helped me get started with the Isabelle system and were always ready to help me solve my daily problems.

Further, I thank the (not yet mentioned) members and guests of the Isabelle working group and office mates Christine Röckl, John Harrison, Wolfgang Naraschewski, Stefan Berghofer, Bernd Grobauer, Gertrud Bauer, Gerwin Klein, Sebastian Skalberg, Martin Strecker, Norbert Schirmer, Raman Ramanujam and Giampaolo Bella as well as Manfred Broy and the remaining colleagues and staff members of our chair for the wonderful working atmosphere.

Very special thanks to the Ph.D. Program "Logic in Computer Science" for generously funding my first three years and providing an excellent environment for the realization of my work.

Finally, I would like to thank my parents Alejandro and Leonor and my friends in Germany and in Spain for their love and support during these four years.

Contents

Chapter 1

Introduction

1.1 Motivation

Parallel programs find applications in many different areas. They offer the possibility of computing data much faster than sequential programs, which is important in applications such as weather forecasting, computer visualized brain surgery, etc. They also support real-time applications where a program (alarm system) cannot be halted to let another program run (deallocation of computer memory), or, they simply need to be synchronized (scheduling resources). However, even small parallel programs are difficult to design, and errors are more often the rule than the exception. Failure of such programs can lead to minor disruptions in daily life, like the loss of a printing job, or to endangering human lives.

Due to their complexity, parallel programs should always be proved correct. This means proving formally that a program performs its intended task for any possible input. This is not easy. Some techniques are based on testing or reasoning about the behavior of possible executions. These techniques help to find bugs and improve efficiency, but do not provide a reliable proof of correctness. In general, it is impossible to verify designs exhaustively by such methods, simply because the number of possible executions is too large (or even infinite). Especially for programs used in safety-critical situations, we need formal verification methods to prove that a program *always* satisfies its specification.

The first formal method for verifying parallel programs was invented already in 1975 by Susan Owicki and David Gries [Owicki, 1975, Owicki and Gries, 1976a, Owicki and Gries, 1976b] and Leslie Lamport [Lamport, 1977]. It was the starting point for further developments like the compositional rely-guarantee method for shared-variable parallelism [Jones, 1981, Jones, 1983] and the assumption-commitment method for synchronous message passing [Misra and Chandy, 1981, Apt *et al.*, 1980, Levin and Gries, 1981].

This dissertation presents the first formalization of the Owicki-Gries and the rely-guarantee methods for correctness proofs of imperative parallel programs with shared variables in a theorem prover. The formalizations have been successfully applied to the verification of several (parameterized and non-parameterized) programs like mutual exclusion and garbage collection algorithms.

1.2 Shared-Variable Parallel Programs

By parallel programs we understand a collection of sequential processes which run concurrently and cooperate in accomplishing some task. This cooperation is possible through the sharing of objects. Depending on the nature of these shared objects we distinguish between shared-variable and message-passing parallelism. In the former, the different processes have access to *shared* memory cells. The latter is used when each process has its own local memory and communicates with other processes by sending messages on *shared* channels. The present work deals with parallel programs that communicate via shared variables only.

1.3 Hoare Logic for Parallel Programs

Verification means proving formally that a program satisfies its specification, which should first be written in some logical language. The approach used in this work is based on the axiomatic method initiated by Hoare for sequential programs and extended later by various researchers to parallel programs. Hoare-like techniques are based on proving that a given program together with its specification can be derived from a system of axioms and inference rules which are syntax oriented, i.e. the proof is carried out on the program's text [Hoare, 1969, Apt, 1981b].

The Owicki-Gries proof system represents the first and probably the simplest extension of Hoare logic to parallel programs with shared-variable concurrency. It provides a methodology for breaking down correctness proofs into simpler pieces. First, the sequential components of the program are annotated with suitable assertions (resulting in so-called proof outlines). Then, the proof reduces to showing that the annotation of each component is correct, and that each assertion of an annotation is invariant under the execution of the actions of the other components (so-called interference freedom of proof outlines).

The main drawback of the Owicki-Gries method is that it is not compositional. To perform the interference-freedom tests for some component we require information about the implementation of all other components. A compositional proof method, however, should be able to infer the specification of the system from the specification of the components without knowing anything about their internal representation.

The idea leading to a compositional proof method for parallel programs is to enrich the specification of each component with additional information about the interaction with the environment during execution. Such a proof method for shared-variable concurrency was first proposed by Cliff Jones [Jones, 1981, Jones, 1983]. A complete version of this system was later designed by [Stølen, 1990].

Jones extended the traditional Hoare pre- and postcondition specification of sequential programs with two new predicates: a *rely* condition specifying what the component expects from the environment, and a *guarantee* condition expressing the task performed by that component, and how this task may influence the environment. These two conditions can be formulated independently of the actual implementation of the components. Then, the verification process consists of proving certain relations among the rely and guarantee conditions of all components (and possibly of an overall environment). As a result, the proof rule for parallel composition can be formulated in terms of the specifications without any need for additional information about the implementation of the components, i.e. the resulting proof method is compositional.

The main improvements of the rely-guarantee method over Owicki-Gries are:

1. The complexity of the verification process grows linearly with the number of components, whereas in the Owicki-Gries method the number of proof obligations grows exponentially.

2. It allows verification of *open systems*, i.e. systems whose interaction with the environment can be specified without knowing the precise implementation of the environment. This makes the method suitable for top-down design. In contrast, the Owicki-Gries method only works for verifying *closed systems*, where the environment is fully characterized in terms of concretely known processes.

Although the rely-guarantee method represents an important step forward in the methodology of program verification, it does not make the Owicki-Gries method obsolete. In general, when it comes to verifying an algorithm, the main concern is finding a proof. In this sense, non-compositional methods are sometimes more successful than compositional ones. This is the case for systems based on shared-variable concurrency like, for example, mutual exclusion algorithms, where processes require reading from and writing to the same shared variable. In general, programs defined using invariants which cannot be easily expressed as a conjunction of local predicates [Chandy and Misra, 1984, Francez and Rodeh, 1980] are difficult for compositional verification methods. There are very few examples of non-trivial shared-variable programs which have been verified using compositional methods [Stølen, 1990, de Boer *et al.*, 1997]. In contrast, non-compositional methods have been quite successful [Gries, 1997, Prensa Nieto and Esparza, 2000, Feijen and van Gasteren, 1999] (see [de Roever *et al.*, 2000] for a discussion on this topic).

Since proving the correctness of parallel programs is indeed difficult, each particular program should be tackled with the most suitable technique. Therefore, it is advantageous to have both compositional and non-compositional systems available.

1.4 Parameterized Parallel Programs

Parameterized parallel programs have become a very important subject of research in the area of computer-aided verification. These are programs which are defined generically, depending in a regular way on a parameter that represents the number of parallel processes. Many interesting programs are of this form, for example, mutual exclusion algorithms for an arbitrary number of processes wanting to use a common resource, or, a garbage collector that interacts with an arbitrary number of user processes. The goal is to verify such systems uniformly, i.e. prove by a *single* proof that the system is correct for any value of the parameter.

The Hoare-like methods for parallel programs found in the literature present systems of rules where the rule for the parallel constructor allows us to derive the composition of some fixed number of processes. In most cases, the parallel constructor is a binary operator [Xu *et al.*, 1997, Stirling, 1988] and programs with more than two processes have to be verified by repeatedly using the rule. Other systems present the parallel composition rule for a fixed number n of known processes [Owicki and Gries, 1976a, Apt, 1981a].

In this thesis, we present a generalization of the parallel composition rule so that parameterized programs can be directly verified in the system. This is achieved by modeling the parallel constructor such that its argument is a list of component programs. Then, the length of the list can be fixed or left as a parameter. With the resulting system, we can derive parameterized parallel programs in one go, i.e. by a single derivation. Several examples have been verified using both the compositional and the non-compositional methods formalized in this thesis. In these examples, the assertions used to obtain a valid derivation in the system are, like the program instructions, parameterized in the number of components and in the particular index of each component.

This led us to the question whether it is always possible to find such assertions, i.e. whether the systems are complete for verifying parameterized parallel programs. Completeness results for the standard systems, where parallel programs have a fixed number of components, have already been presented in [Owicki, 1975, Apt, 1981a] for the Owicki-Gries system and in [Xu *et al.*, 1997] for the rely-guarantee system. These results ensure that for any correct parallel program with a fixed number of components we can find a derivation in the respective system. However, they do not solve the problem for the parameterized case. In this thesis, we present proofs of completeness of the systems for parameterized programs as a natural extension of the known completeness results. These proofs have been carried out in an informal pencil-and-paper style, i.e. not formalized in the theorem prover.

1.5 Need for Machine-Supported Verification

Whichever method we select for an application, the main difficulty lies in finding assertions that formally express the conditions of the specification and yield a derivation from the corresponding system of rules.

For the Owicki-Gries method we need to find interference free intermediate assertions that lead to the expected results. For the rely-guarantee method, we need appropriate rely and guarantee conditions. These assertions are usually difficult to find and have to be changed and tuned many times. Each time the verification process has to be essentially restarted.

It is possible to do the proof by hand. However, this is a long, tedious and error-prone process. Consider for example the Owicki-Gries method. The number of interference-freedom tests needed is $O(k^n)$, where n is the number of sequential components, and k is the maximal number of lines of a component. This makes a complete pencil-and-paper proof very laborious, even for small examples. For this reason, many of the interference-freedom proofs, which tend to be very simple, are usually omitted. A fact that increases the possibility of a mistake.

With the rely-guarantee method the situation is not so critical because the number of conditions to be checked is considerably smaller. Nevertheless, the proofs involved in verification are very detailed and often just boring routine work. When proofs are done by hand, one tends to spend too much time checking rather simple proof steps over and over again. Therefore it is desirable to have the help of computers that automate the process and ensure that no mistakes are made.

1.5.1 Theorem Proving vs. Model Checking

There are two major approaches used for mechanizing verification: theorem proving and model checking. Theorem proving is based on using a specific deductive system and performing a formal proof in the mathematical sense. Model checking techniques are based on decision or semi-decision procedures that "check" whether a program satisfies its specification by basically exploring the possible states exhaustively. Model checking has the advantage of being essentially automatic whereas general theorem proving requires interactive input from the user. On the other hand, verification with theorem provers can be fully general whereas model checking is still only applicable to a limited class of programs.

Initially, model-checking was restricted to finite-state systems of moderate size, but thanks to the development of techniques that improve efficiency it is now possible to tackle surprisingly large examples. Unfortunately, for infinite-state systems, even the theoretical possibility of exploring the state space disappears.

A program may have an infinite state space because it operates on data structures from a potentially infinite domain (integers, queues, etc.) or because it has an infinite control part (parallel programs parameterized in the number of components). Theoretical results [Apt and Kozen, 1986] even show that the verification of parameterized parallel programs is undecidable. Nevertheless, recent work present solutions that extend the applicability of model checking to restricted cases of infinite-state systems of the first kind [Jonsson and Parrow, 1993, Alur and Dill, 1994, Henzinger, 1995], and of the second kind [German and Sistla, 1992, Clarke *et al.*, 1995, Esparza, 1995, Abdulla and Jonsson, 1998]. However, programs which are infinite in both data and control flow are out of reach for the existing model-checking techniques.

The availability of a theorem prover allows us to reason generically about programs without restrictions on their specification. It can deal with unbounded or infinite systems and supports highly expressive specifications of properties. For example, a garbage collection algorithm that manipulates an infinite data structure representing the computer memory and interacts with a parameterized number of mutators can be naturally specified and verified [Prensa Nieto and Esparza, 2000].

In this work a powerful interactive theorem prover, Isabelle, is used to formalize two well-known axiomatic verification methods, prove their soundness and considerably automate their application to real programs. As a result, we obtain a verification tool for general parallel programs where the generation of the verification conditions and the proof of the easy cases is automatic. The user is then able to concentrate only on the most important aspects of the proof.

1.6 Related Work

The idea of embedding an imperative programming language in a theorem prover goes back at least to [Gordon, 1989], who considered the Hoare logic of a simple while-language. Gordon's idea inspired an increasingly active research area. In this section, we give a brief overview. The following list of references is by no means exhaustive.

In the line of sequential language embeddings, [Nipkow, 1996, Nipkow, 1998] formalizes the first chapters of [Winskel, 1993] in Isabelle/HOL (and even finds a mistake in the proof of completeness), [Harrison, 1998] presents a formalization in HOL of Dijkstra's classic [Dijkstra, 1976], [Homeier and Martin, 1996] and [Kleymann, 1998] deal also with recursive procedures, [von Oheimb, 2001] presents an embedding of a Hoare logic for a subset of sequential Java in Isabelle/HOL and [Filliatre, 1999] presents a formalism for the verification of imperative programs in Type Theory in Coq.

For concurrent programming languages we encounter a long list of embeddings. For example, UNITY has been formalized in the Boyer-Moore prover [Goldschlag, 1990], HOL [Andersen *et al.*, 1994], Coq [Heyd and Crégut, 1996], LP [Chetali and Heyd, 1997] and Isabelle [Paulson, 2000, Paulson, 2001]. A related framework, *action systems*, has also been formalized in HOL [Långbacka and von Wright, 1997]. CSP has been treated in HOL [Camillieri, 1990], in Isabelle/HOL [Tej and Wolff, 1997] and in PVS [Dutertre and Schneider, 1997]. CCS has been formalized in HOL [Nesi, 1994]. ACP-style process algebra has also been formalized in PVS [Basten and Hooman, 1999]. TLA is found in HOL [von Wright and Långbacka, 1993], LP [Engberg *et al.*, 1993] and Isabelle/HOL [Kalvala, 1995]. Input/Output Automata embeddings are found in Isabelle/HOL [Müller and Nipkow, 1997, Müller, 1998] and in LP [Søgaard-Andersen *et al.*, 1993]. A formalism based on a *semantical* characterization of compositional verification has been formalized in PVS [Owre *et al.*, 1995]. [Hooman, 1998] presents an embedding of an assertional compositional system for asynchronous communication in PVS. The PVS system combined with model checking techniques [Rushby, 2000, Owre *et al.*, 1996, Shankar, 1996] has been successfully applied to medium-size examples as reported for example in [Hooman, 1995, Shankar, 1998, de Roever *et al.*, 1998].

Surprisingly, it appears that there has been no work on embedding Hoare logics for shared-variable parallelism in any theorem prover.

The Owicki-Gries method marks the beginning of a vast body of literature on proof systems for concurrency which we cannot survey here. The recent book [de Roever *et al.*, 2000] presents a development of state-based verification systems and contains numerous references related to the subject. A second volume of this book is announced [Hooman *et al.*, 2000] which focuses on illustrating through examples the success of compositional techniques in correctness proofs for parallel programs as well as techniques for machine-support in the verification process using PVS.

1.7 Formalization in Isabelle/HOL

For the formalization and proofs presented in this work we use the system Isabelle/HOL. Isabelle [Paulson, 1994] is a generic interactive theorem prover and Isabelle/HOL is its instantiation for higher-order logic, which is very similar to Gordon's HOL system [Gordon and Melham, 1993]. A recent gentle introduction to Isabelle/HOL is [Nipkow and Paulson, 2001].

In this section we briefly describe the aspects of the system required to understand this dissertation. We can divide the work done with Isabelle into two main parts: the formalization of the verification methods and their application to concrete programs.

1.7.1 Formalization of the Verification Methods

When formalizing a programming language in a theorem prover, one has to decide between using a *deep embedding*, where first the (abstract) syntax is represented via an inductive datatype and then a semantics is assigned to it, and a *shallow embedding*, where a term in the language is essentially an abbreviation of its semantics. Deep embeddings are useful when meta-theoretic reasoning (usually by induction over the syntax) is required. Shallow embeddings on the other hand, simplify reasoning about individual programs because one may work directly with the semantics avoiding the extra syntactic level.

We use a combination of both styles that has become quite established [Nipkow, 1998, von Oheimb, 2001]. We formalize as much as possible using a shallow embedding and use a deep embedding only where needed in order to perform the meta-theoretic proofs we are interested in. For our purposes, it suffices to use a deep embedding for the programming language. Assertions, expressions and even assignments are represented semantically, i.e. as functions on *states*. Consequently, the assertion language is not restricted to first-order logic as is customary in Hoare-like frameworks. Any HOL expression, and in particular all the constants defined in the Isabelle/HOL library can be used in assertions, boolean conditions and expressions within a program.

The program syntax is defined in Isabelle/HOL via a **datatype** definition. A free datatype is defined by listing its constructors together with their argument types, separated by '|'. In general it has the form

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ t = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid\ C_m\ \tau_{m1} \ldots \tau_{mk_m}$$

where α_i are distinct type variables, C_i are distinct constructor names and τ_{ij} are types. Type abbreviations in Isabelle are declared by the keyword **types**. They follow the syntax of ML, except that function types are denoted by $\Rightarrow$. Laws about datatypes, such as $C_i \neq C_j$, are automatically included in the simplification tactics for future proofs. An induction principle, namely, structural induction over the constructors of the datatype is also generated with each datatype declaration. To use it in proofs it has to be explicitly invoked. Functions about datatypes are usually defined by primitive recursion. They are introduced by the keyword **primrec**.

Constants are declared with **consts** followed by their name and type, separated by '::'. Non-recursive definitions are declared by the keyword **contsdefs**. The introduced constant and its definition are separated by '$\equiv$'. Sometimes we first declare the constant and introduce the definition later with the keyword **defs**.

The operational semantics of commands is inductively defined via a set of rules. Similarly, the set of correct specifications is defined inductively by a set of axioms and proof rules. Such inductively defined sets represent the least set which is closed under the formation rules. They are declared by the keyword **inductive** followed by the word **intros**. From each inductive definition Isabelle generates the corresponding induction principle, called *rule induction*, which represents the most powerful proof method used in this dissertation. In particular, soundness of the system of rules for program verification is proved by rule induction.

The so-called *inductive cases* proof principle is also automatically generated by the system for any inductive definition. It can be understood as the counterpart of (structural) case distinction on inductively generated elements. Whenever we have an assumption stating that an element belongs to an inductively defined set, we can distinguish on the last rule used for its derivation. As a result, we obtain a subgoal where the given element has been replaced by the corresponding premises of each of the proof rules whose conclusion matches the element. When we speak of case analysis on an inductively generated element we refer to this proof principle.

Statements that we want to prove are preceded by **theorem** or **lemma**. There is no formal difference between them; we use one or the other depending on the importance we attach to the stated proposition. Proofs are done by applying *tactics* to the stated goals. The application of a tactic is preceded by the keyword **apply**. The basic tactics are based on *resolution*, i.e. by applying inference rules (backwards or forwards) in a natural deduction style, and *rewriting*, i.e. by applying (conditional) directed equalities. As a result, goals are reduced to simpler subgoals until they become trivial. When all subgoals are solved the proposition is proven and stored under some name given by the user.

Some tactics are based on natural deduction (forward and back-chaining of rules) where search with backtracking is automated using the so-called *classical reasoner*. Other tactics, called *simplifiers*, compose rewriting steps. More powerful tactics (like *auto*) combine both systems and are able to automatically prove complicated goals. Tactics may also be combined using control structures called *tacticals*.

In an interactive theorem prover like Isabelle, if a statement cannot be proved automatically, the user is able to direct the proof by explicitly using induction, case distinction, instantiating variables and, in general, giving hints to the prover whenever automatic tactics do not succeed. Very often, automatic tools do succeed if they are supplied with the suitable auxiliary lemmas, which can be often obtained from the Isabelle library or have to be previously proven by the user.

1.7.2 Application to Concrete Programs

The relation between the program syntax and its semantics need only be studied once in the proof of soundness of the system. Once this is done, we can forget about the semantics and just use the system of rules for the verification of programs.

In order to make the verification task easier, we enrich the formalization with two additional features: the definition of a familiar concrete syntax for the programming language, and a tactic that, given a program specification, automatically generates the verification conditions.

Isabelle offers several facilities to define concrete (or external) syntax. It is possible to declare mixfix syntax notation for types and constants. The new notation may contain mathematical symbols and user-defined precedences. They can be directly given with the constant or type declaration (by writing the mixfix notation in parenthesis) or by defining new syntactic constants under the keyword **syntax** and putting the corresponding translation equations into the internal (abstract) syntax under the keyword **translations**. The translation equations are directed. The direction from left to right is represented by the arrows $\rightharpoonup$, and the translation in both directions by $\rightleftharpoons$.

When the transformations cannot easily be done via translations, Isabelle offers the possibility of defining ML programs that perform the translations from concrete syntax into abstract syntax (**parse_translation**), and vice versa (**print_translation**). These facilities are used to obtain an alternative syntax that allows us to write programs and assertions essentially like they are found in the literature.

The correctness of a program specification depends upon the validity of certain conditions called *verification condition* (also called *proof obligations*). These conditions are pure higher-order logic predicates with no mention of the programming language. Their validity is thus proven using standard Isabelle proving techniques.

For the automatic generation of the verification conditions, we define a so-called *verification conditions generator (vcg)* as an Isabelle tactic. Isabelle allows the user to construct new tactics by programming them in ML as combinations of existing ones. Without this tactic, the verification of the larger examples presented in this thesis would have been unbearably tedious.

1.8 The Standard Isabelle/HOL Library

The formalization uses some types and constants defined in the standard Isabelle/HOL library. We briefly present the most frequently used. Others will be explained when needed in the subsequent chapters.

The product type $\alpha \times \beta$ comes with the projection functions *fst* and *snd*. Tuples are pairs nested to the right, e.g. $(a,\ b,\ c) = (a,\ (b,\ c))$. They may also be used as patterns like in $\lambda(x, y).\ f\ x\ y$.

List notation is similar to ML (e.g. @ is 'append') except that the 'cons' operation is denoted by # (instead of : :). The ith component of a list xs is written $xs\,!\,i$, where the first element has the index 0, i.e. $xs!0$, also defined as the head of the list, *hd* xs. The rest of the list (or tail) can be represented by the function *tl* xs. *last* xs represents the last element of a non-empty list. The syntax $xs[i := x]$ denotes xs with the ith component replaced by x. The functional *map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *list* $\Rightarrow \beta$ *list* applies a function to all elements of a list. The function *length* :: α *list* $\Rightarrow$ *nat* returns the length of a list. The conversion function *set* :: α *list* $\Rightarrow \alpha$ *set* builds a set from the elements of a list. The function *filter* :: $(\alpha \Rightarrow$ *bool*$) \Rightarrow \alpha$ *list* $\Rightarrow \alpha$ *list*, returns the list of elements formed from the elements of a given list for which a given predicate holds.

The datatype α *option* = *None* | *Some* α is frequently used to add a distinguished element to some existing type. It comes with the function *the* :: α *option* $\Rightarrow \alpha$ such that *the (Some x) = x*.

Set comprehension syntax is $\{x.\ Px\}$ expressing the set of all elements that satisfy the predicate P. This notation is also available for tuples $\{(x, y, z).\ Pxyz\}$. A more general syntax for sets is $\{e\vec{x} \mid \vec{x}.\ P\vec{x}\}$ which abbreviates the set $\{u.\ \exists\vec{x}.\ u = e\vec{x} \wedge P\vec{x}\}$. The image of a set A under a function f is denoted by f ' A and is predefined in the Isabelle library as $\{f\ x \mid x.\ x \in A\}$. The complement of a set A is denoted by $-A$.

The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ represents an implication with assumptions $A_1, \ldots, A_n$ and conclusion A. It is also important to distinguish between the object implication '$\longrightarrow$' and the meta-implication '$\Longrightarrow$'. The first one is a normal implication as known from mathematics and the second one separates assumptions from conclusions in proofs. In other words, if we state the goal $a \longrightarrow b$ in Isabelle, then we want to prove the proposition $a \longrightarrow b$ from the empty set of assumptions. However, if we state $a \Longrightarrow b$, then we want to prove b by assuming a. The first goal can be reduced to the second one by applying the deduction rule $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$ backwards.

1.9 Presentation Style

Isabelle provides tools for the automatic generation of LaTeX documents from Isabelle theories. The formal content of this dissertation has been written with this system. Hence, all definitions and theorems presented are part of actual Isabelle theories, where the side explanations have been inserted in special "text" environments that are ignored when the theory is being processed by Isabelle. This procedure ensures the consistency of the information presented. However, it forces the presentation to be bottom-up (as in the original formal theories) where probably a top-down explanation of the contents would be more appropriate. Nevertheless, we manage to avoid showing unnecessary details of the formalization (like user-defined preferences in syntax declarations) and more technical parts that are not relevant for the general understanding (like the syntax transformation functions or the tactics programmed in ML) are placed in the appendix.

The proofs done in this work follow the "traditional" tactic style, resulting in so-called "proof scripts", which are not easily readable for non-experienced users. Now there is an alternative based on a more developed proof language that allows us to write proofs basically as they are found in mathematics books. This new system is called Isar (Intelligible semi-automated reasoning) and has been developed by Markus Wenzel [Wenzel, 2001a]. We have adopted the definition language of Isar but have maintained the tactic-style proofs (for historical reasons). Thus, we do not show the mechanical proofs in this thesis, but simply state the lemmas and explain their proofs informally in the text. Longer proofs are announced by **Proof** and finished with □. The complete Isabelle theories and proof scripts can be obtained from `http://isabelle.in.tum.de/hoare-parallel/`.

1.10 Overview

Chapter 2 presents the formalization of the Owicki-Gries method and its application to some typical examples.

Chapter 3 is devoted to the main case study, namely, the verification of two parallel garbage collection algorithms, the second one parametric in the number of mutators.

Chapter 4 presents the formalization of the rely-guarantee method and its application to some typical examples.

Chapter 5 is devoted to the completeness of both the Owicki-Gries and rely-guarantee systems for parameterized parallel programs.

Chapter 6 summarizes the main results and gives suggestions for further work.

All results except for the completeness theorems of chapter 5 have been obtained using the theorem prover Isabelle/HOL.

Part of the materials contained in this thesis have been previously published in [Nipkow and Prensa Nieto, 1999], [Prensa Nieto and Esparza, 2000] and [Prensa Nieto, 2001].

Chapter 2

The Owicki-Gries Method in Isabelle/HOL

In this chapter we present the first formalization in a theorem prover of the Owicki-Gries method. First published by Susan Owicki in her Ph. D. thesis under the supervision of David Gries [Owicki, 1975], this method is widely accepted as the most fundamental methodology for correctness proofs of shared-variable concurrency.

Our formalization closely follows the description in [Apt and Olderog, 1991]. Thus, the present work can also be seen as an exercise in formalizing textbooks on programming language semantics. Yet, the search for a suitable and efficient adaptation in the theorem prover yields some improvements over the original presentation. Especially interesting is the parametric nature of the parallel composition rule (allowing verification of parameterized parallel programs directly in the system) and the soundness proof (because it does not explicitly mention program locations).

The different parts of the formalization are introduced following the usual steps required by a formal system. First, the abstract syntax of the programming language presents a simple while-language with concurrent execution of commands and synchronization via an await-command. Then, the operational semantics and the proof system are inductively defined as sets of rules. The soundness of the latter w.r.t. the former represents the main meta-theoretical result of the formalization.

Our interest is focused on the practical application. Therefore we prove soundness but not completeness of the proof system. Instead, we provide an automatic procedure for the generation of the verification conditions and define a familiar concrete syntax for writing programs. Finally, some typical examples illustrate the applicability of the formalization. Furthermore, the verification of several schematic programs, where the number of parallel components is a parameter, shows that this embedding is more than just a verification condition generator. The use of a theorem prover allows us to tackle problems outside the range of fully automatic methods like model checking.

In the next chapter the method is applied to two garbage collection algorithms, the second one parametric in the number of mutators. These nontrivial case studies demonstrate the success of the approach, mainly due to the high degree of automation.

2.1 Abstract Syntax

We follow [Apt and Olderog, 1991] in stratifying the language. Only top-level parallelism is allowed, i.e. the parallel operator ($\|$) must not be nested. Hence, each c_i in $c_1 \parallel \ldots \parallel c_n$ is a sequential command, called a *(sequential) component* of the parallel composition. Nevertheless, parallelism may occur within sequential composition, conditional statements and while-loops.

The third sublanguage in this stratification is the one used in the bodies of await-commands. They are called *atomic programs* because they are executed atomically, i.e. without interruption from other components. Summarizing, the programming language combines the following three layers:

Parallel commands include parallel composition of component programs as a construct. Component programs are purely sequential programs, thus nested parallelism is excluded.

Component commands represent the language of the programs appearing within a parallel composition and can be synchronized via an await-command. They have to be annotated with assertions before each command, thus, they are also called *annotated commands*.

Atomic commands are used for the bodies of await-statements, which are executed atomically.

We illustrate by the following example the scope of each layer. The delimiters **cobegin-coend** enclose a list of programs that are to be executed in parallel:

> $x := 0;$
> **cobegin**
> $\{x = 0\}$ **await** *True* **then** $x := x + 1;\ x := x + 1$ **end** $\{True\}$
> $\|$
> $\{True\}\ x := 0\ \{x = 0\ \vee x = 1\ \vee x = 2\}$
> **coend**

The whole is a *parallel* command (non-annotated)consisting of the sequential composition of an assignment (x:=0) and a parallel composition of two component (annotated) commands. The first component consists of an await-statement whose body is an atomic command (non-annotated). In HOL, there are two ways to encode this stratification:

1. Define the type of all programs and require well-formedness predicates for each sublanguage, or

2. Define the syntax in layers with different types.

We have chosen a combination of both that simplifies statements and proofs about the language. Component programs and parallel programs are defined in different layers, whereas atomic programs are defined as a sublanguage of parallel programs via a simple well-formedness predicate.

Although a number of constructs appear duplicated in both the parallel and component layers, they differ in that the latter attaches annotations to them. Proofs about those constructs may have to be duplicated but this duplication is quite mechanical.

Following the established combination of shallow and deep embedding we start by defining the parameterized type abbreviations:

types
  $\alpha\ bexp = \alpha\ set$
  $\alpha\ assn = \alpha\ set$

representing both assertions and boolean expressions as sets (of states). The α stands for the state of a program, which is a parameter of the program type. The reason for this formalization of the state will be explained in §2.7.

The three levels of the language depend on each other. An await-command is a constructor of the language of component programs but its body must be atomic. Similarly, a parallel construct is not part of the component's language but its arguments are component programs.

Atomic commands represent the simpler layer in this hierarchy. Its constructors are also used in parallel commands so that atomic programs can be defined as a sublanguage of parallel programs. Hence, only two datatypes are required: α *ann-com* for annotated sequential programs and α *com* for atomic and parallel programs. Datatypes that depend on each other are called *mutually recursive*. They are defined in HOL under a single datatype declaration joined via the keyword **and**:

datatype α *ann-com* =
  AnnBasic (α *assn*) ($\alpha \Rightarrow \alpha$)
 | *AnnSeq* (α *ann-com*) (α *ann-com*)
 | $\mathit{AnnCond}_1$ (α *assn*) (α *bexp*) (α *ann-com*) (α *ann-com*)
 | $\mathit{AnnCond}_2$ (α *assn*) (α *bexp*) (α *ann-com*)
 | *AnnWhile* (α *assn*) (α *bexp*) (α *assn*) (α *ann-com*)
 | *AnnAwait* (α *assn*) (α *bexp*) (α *com*)
and α *com* =
  Parallel (α *ann-com option* $\times$ α *assn*) *list*
 | *Basic* ($\alpha \Rightarrow \alpha$)
 | *Seq* (α *com*) (α *com*)
 | *Cond* (α *bexp*) (α *com*) (α *com*)
 | *While* (α *bexp*) (α *assn*) (α *com*)

2.1.1 Component Programs

The language of component programs has the type α *ann-com*. It is a standard sequential while-language augmented with a synchronization construct (*AnnAwait*). It departs from the usual presentation of the language by the inclusion of assertions directly in the syntax: every construct, apart from sequential composition, is annotated with a precondition, and the loop is also annotated with an invariant. Due to this special presentation they are called *annotated commands*. We emphasize that these assertions are merely annotations and do not change the semantics of the language. Next, we discuss the different constructors:

AnnBasic represents a basic atomic state transformation, for example an assignment, a multiple assignment, or even any non-constructive specification. Concrete syntax for single assignments of the form x:=e, where x is a program variable and e is an expression of the corresponding type, is also available (cf. §2.7).

AnnSeq is the sequential composition of commands.

AnnCond$_1$ is the standard conditional. Both subprograms are themselves annotated commands, thus preceded by a precondition, subject to the so called *interference freedom* test in the verification process.

AnnCond$_2$ is the conditional without else-branch. In some cases, it is convenient to ignore the else-branch altogether because its precondition is bound to fail at the interference freedom test.

AnnWhile is the loop, annotated with an invariant.

AnnAwait is the synchronization construct. Notice that its body is of type α *com*.

It might seem more natural to define *AnnCond*$_2$ as a special case of the constructor *AnnCond*$_1$, namely *AnnCond*$_1$ r b c (*AnnBasic* p *id*), where *id* represents the identity transformation. This, however, would still require proving interference freedom for the assertion p which must, in general, include the clause $\neg b$. Sometimes, e.g. in §3.4.1, this is not possible and we prefer to directly ignore the else-part. Another possible way of avoiding this assertion is to consider *AnnCond*$_2$ as an abbreviation of *AnnCond*$_1$ r b c (*AnnAwait* r *True* (*Cond* b c (*Basic* *id*))). This way we also avoid proving that $\neg b$ is interference free, but it is an unnecessarily complicated solution.

The meaning of await-commands is quite intuitive to understand. Imagine an execution of a parallel composition of programs where one component intends to execute the statement *AnnAwait* r b c. If b evaluates to *True*, then c is executed as an atomic region. If b evaluates to *false*, the component becomes blocked. In sequential programs this behavior does not make any sense, but in the case of parallel programs it means that other components can take over the execution. If eventually b becomes true, the blocked component can resume its execution. Otherwise, it remains blocked forever. Programs with this construct may end in a *deadlock*. This happens when some component of a parallel program is blocked and there are no other active components. A component is *active* if it is neither blocked nor finished.

To reason about parallel programs with shared variables we need to reason about each atomic step taken in the computations of its components. To this end, proofs of component programs are presented in the form of *proof outlines*, i.e. interleaved with assertions at appropriate places. Furthermore, we directly present them as a special case of proof outlines called *standard proof outlines* obtained by minimizing the number of annotations. Each command c is then preceded by an assertion, *pre* c, and apart from these and loop invariants there are no other assertions[1]. These annotations describe (a subset of) the set of states that are reachable at each point of control.

For purely sequential programs such a presentation is not necessary. The intermediate annotations can be derived as the weakest precondition from the postcondition and loop invariants. The so computed assertions invariably hold at their respective locations since no other action can modify the expected results. In contrast, a component in a parallel program has the ability to modify shared variables, endangering the task of other components. For this reason we explicitly annotate each point of control with an assertion, whose invariance under the actions in the other parallel components can be checked.

To obtain this special presentation we include the precondition in the syntax of component programs. Moreover, it turns out that for proof-theoretic reasons it is very helpful to define the semantics of the language directly on annotated commands. The precondition of each annotated command is extracted by the function *pre*. Its definition is:

consts
  $pre :: \alpha\ ann\text{-}com \Rightarrow \alpha\ assn$
primrec
  $pre\ (AnnBasic\ r\ f) = r$
  $pre\ (AnnSeq\ c_1\ c_2) = pre\ c_1$
  $pre\ (AnnCond_1\ r\ b\ c_1\ c_2) = r$
  $pre\ (AnnCond_2\ r\ b\ c) = r$
  $pre\ (AnnWhile\ r\ b\ i\ c) = r$
  $pre\ (AnnAwait\ r\ b\ c) = r$

2.1.2 Atomic and Parallel Programs

Atomic and parallel commands are similar enough to both be represented by the same datatype ($\alpha\ com$) together with a simple well-formedness predicate characterizing atomic programs. We could define them in two different datatypes, but this would make the specification of the language unnecessarily long and proofs about (otherwise) identical constructors would have to be duplicated. By minimizing the number of constructors in the language we obtain a clearer and shorter presentation of the theories and proofs.

Atomic commands form the body of await-statements which are executed as atomic regions, i.e. its activation cannot be interrupted by the other components. Hence, they behave as pure sequential programs. On the other

[1]Non-standard proof outlines admit two assertions after each other provided that the second one is a logical consequence of the first one.

hand, parallel commands are themselves not executed in parallel, i.e. nested parallelism is not allowed. However, if c_1 and c_2 are both parallel programs, they can be sequentially composed ($\mathit{Seq}\ c_1\ c_2$) or appear in conditional ($\mathit{Cond}\ b\ c_1\ c_2$) statements and loop constructions ($\mathit{While}\ b\ i\ c_1$).

The only kind of commands in this layered language that is executed in parallel with others are the annotated commands.

Consequently, atomic programs and programs containing parallel programs are similar in the sense that they are both sequential. This is characterized by two important aspects:

1. They do not contain await-statements, which are only meaningful in a parallel context. To ensure termination, it is also usual to forbid while-commands inside atomic regions. However, this condition is not necessary for the soundness of the system and so we leave it out.

2. They do not contain intermediate annotations. For purely sequential commands there is no need to record a proof outline to be checked for interference freedom.

The *Parallel* constructor encloses a list of pairs $(c,\ q)$, where c is a sequential command or the empty program if the execution has terminated, and q is the postcondition (remember that the precondition is already part of the annotated c). Strictly speaking it is not necessary to include the postcondition, but it simplifies program verification.

Although each component consists of a pair, they can be seen as Hoare triples. The three elements are the precondition, which can be extracted from the command, followed by the program itself (ignoring the precondition), and finally the postcondition.

The remaining commands are almost like their namesakes in the sequential layer, but with a slightly different concrete syntax for sequential composition, to avoid confusion.

Nevertheless, atomic commands do not contain parallel constructs. We introduce a well-formedness predicate that characterizes the subset of programs of $\alpha\ \mathit{com}$ that may appear inside atomic regions:

consts $\mathit{atom\text{-}com} :: \alpha\ \mathit{com} \Rightarrow \mathit{bool}$
primrec
  $\mathit{atom\text{-}com}\ (\mathit{Parallel}\ \mathit{Ts}) = \mathit{False}$
  $\mathit{atom\text{-}com}\ (\mathit{Basic}\ f) = \mathit{True}$
  $\mathit{atom\text{-}com}\ (\mathit{Seq}\ c_1\ c_2) = (\mathit{atom\text{-}com}\ c_1 \wedge \mathit{atom\text{-}com}\ c_2)$
  $\mathit{atom\text{-}com}\ (\mathit{Cond}\ b\ c_1\ c_2) = (\mathit{atom\text{-}com}\ c_1 \wedge \mathit{atom\text{-}com}\ c_2)$
  $\mathit{atom\text{-}com}\ (\mathit{While}\ b\ i\ c) = \mathit{atom\text{-}com}\ c$

If desired while-commands could also be excluded by writing

$$\textit{atom-com}\ (\textit{While}\ b\ i\ c) = \textit{False}$$

instead.

2.2 Operational Semantics

The semantics defines the input/output behavior of programs, i.e. given a program c, its semantics *Sem* c defines a mapping from (initial) states to (final) states. There are two classical ways of defining this mapping:

A denotational semantics [Scott and Strachey, 1971, Gordon, 1979] defines *Sem* c by induction on the structure of c, as a partial function on states. In particular, fixed point techniques are used to deal with recursion.

An operational semantics [Hennessy and Plotkin, 1979, Plotkin, 1981] defines first a transition relation between so-called *configurations* and then defines *Sem* c using this relation.

Although the denotational style is more abstract and can theoretically handle all programming languages, it becomes very complicated for parallel programs. In contrast, the operational semantics remains simple and is thus preferred for assigning meaning to parallel constructors.

2.2.1 The Transition Relation

The transition relation used to define the operational semantics is inductively defined by a set of axioms and rules about transitions (or steps) between configurations. A *configuration* is a pair of a program fragment and a state, where a program fragment is either an atomic command or, if execution has come to an end, the empty program. Each transition is regarded as one step in the computation. For example, $(c,\ s)\ {-1\!\!\rightarrow}\ (c',\ s')$ means that the execution of one instruction in c from state s leads to the configuration consisting of a command c' and a state s' from which execution continues.

Basically, we need to define axioms and rules for all possible transition steps. Since the programming language constructs are defined in two different layers, two kinds of transitions are defined: *transition*, for steps of commands of type α *com* and, *ann-transition*, for steps of commands with type α *ann-com*.

In order to define the rules we need some way to represent the fact that the command is empty. The language of component programs does not contain the empty program. Adjoining a new element to a type is naturally modeled by the standard Isabelle/HOL datatype α *option* = *None* | *Some* α. In this case, *None* represents the empty program. Otherwise, the command is wrapped up by the *Some* constructor. To abbreviate we define a new type for optional annotated commands:

types α *ann-com-op* = α *ann-com option*

A new type abbreviation for an optional annotated command followed by its postcondition stands for the type of each component in the list of a parallel composition constructor. It can be seen as a triple since the precondition is part of the command's type:

types α *ann-triple-op* = (α *ann-com-op* $\times$ α *assn*)

Two selector functions extract the command part and the postcondition:

consts *com* :: α *ann-triple-op* $\Rightarrow$ α *ann-com-op*
primrec *com* $(c,\ q) = c$

consts *post* :: α *ann-triple-op* $\Rightarrow$ α *assn*
primrec *post* $(c,\ q) = q$

Equations defining a primitive recursive function are automatically added to the simplifier.

In the language of atomic and parallel programs we can consider the parallel composition with the empty list as argument as the equivalent of the empty program. This way we avoid the option type. The execution of a parallel composition terminates when all components do, i.e. when all components are *None*. The following predicate characterizes a terminated parallel composition:

constdefs
  All-None :: α *ann-triple-op list* $\Rightarrow$ *bool*
  All-None Ts $\equiv \forall (c,\ q) \in$ *set Ts*. c = *None*

Now we can define the transition relations. They are inductively defined as sets of relations between configurations:

consts
  $\mathit{ann\text{-}transition} :: ((\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha) \times (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha))\ \mathit{set}$
  $\mathit{transition} :: ((\alpha\ \mathit{com} \times \alpha) \times (\alpha\ \mathit{com} \times \alpha))\ \mathit{set}$

Concrete syntax for a transition step as well as for its n-fold iteration and the reflexive transitive closure is provided via infix syntax annotations

syntax
  $\text{-}\mathit{ann\text{-}transition} :: (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha) \Rightarrow (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha) \Rightarrow \mathit{bool}$
  $(\text{-}\ {-1\!\rightarrow}\ \text{-})$
  $\text{-}\mathit{ann\text{-}transition\text{-}n} :: (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha) \Rightarrow \mathit{nat} \Rightarrow (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha)$
  $\Rightarrow \mathit{bool}\ \ (\text{-}\ {-\text{-}\!\rightarrow}\ \text{-})$
  $\text{-}\mathit{ann\text{-}transition\text{-}*} :: (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha) \Rightarrow (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha) \Rightarrow \mathit{bool}$
  $(\text{-}\ {-*\!\rightarrow}\ \text{-})$

  $\text{-}\mathit{transition} :: (\alpha\ \mathit{com} \times \alpha) \Rightarrow (\alpha\ \mathit{com} \times \alpha) \Rightarrow \mathit{bool}\qquad (\text{-}\ {-P1\!\rightarrow}\ \text{-})$
  $\text{-}\mathit{transition\text{-}n} :: (\alpha\ \mathit{com} \times \alpha) \Rightarrow \mathit{nat} \Rightarrow (\alpha\ \mathit{com} \times \alpha) \Rightarrow \mathit{bool}\quad (\text{-}\ {-P\text{-}\!\rightarrow}\ \text{-})$
  $\text{-}\mathit{transition\text{-}*} :: (\alpha\ \mathit{com} \times \alpha) \Rightarrow (\alpha\ \mathit{com} \times \alpha) \Rightarrow \mathit{bool}\qquad (\text{-}\ {-P*\!\rightarrow}\ \text{-})$

The corresponding syntax translations are:

translations
  $\mathit{con}_0\ {-1\!\rightarrow}\ \mathit{con}_1 \rightleftharpoons (\mathit{con}_0,\ \mathit{con}_1) \in \mathit{ann\text{-}transition}$
  $\mathit{con}_0\ {-n\!\rightarrow}\ \mathit{con}_1 \rightleftharpoons (\mathit{con}_0,\ \mathit{con}_1) \in \mathit{ann\text{-}transition}\hat{\ }n$
  $\mathit{con}_0\ {-*\!\rightarrow}\ \mathit{con}_1 \rightleftharpoons (\mathit{con}_0,\ \mathit{con}_1) \in \mathit{ann\text{-}transition}^*$

  $\mathit{con}_0\ {-P1\!\rightarrow}\ \mathit{con}_1 \rightleftharpoons (\mathit{con}_0,\ \mathit{con}_1) \in \mathit{transition}$
  $\mathit{con}_0\ {-Pn\!\rightarrow}\ \mathit{con}_1 \rightleftharpoons (\mathit{con}_0,\ \mathit{con}_1) \in \mathit{transition}\hat{\ }n$
  $\mathit{con}_0\ {-P*\!\rightarrow}\ \mathit{con}_1 \rightleftharpoons (\mathit{con}_0,\ \mathit{con}_1) \in \mathit{transition}^*$

The last two arrows are syntactic sugar for the n-fold and the * postfix operators which are part of Isabelle/HOL's theory of relations.

The two kinds of transitions defining the semantics depend on each other. Thus, the rules for both systems are defined simultaneously and atomic programs and parallel programs "share" the rules for the common constructors.

The transition rules defined below are also called small-step rules and the semantics they define is called small-step semantics because it describes the execution of programs step by step. (In contrast to the so-called big-step semantics, where the rules represent transitions that may correspond to several steps in the execution of the program.)

inductive *ann-transition transition*
intros
AnnBasic: $(Some\ (AnnBasic\ r\ f),\ s)\ {-}1{\rightarrow}\ (None,\ f\ s)$

AnnSeq1: $(Some\ c_0,\ s)\ {-}1{\rightarrow}\ (None,\ t) \Longrightarrow$
$(Some\ (AnnSeq\ c_0\ c_1),\ s)\ {-}1{\rightarrow}\ (Some\ c_1,\ t)$
AnnSeq2: $(Some\ c_0,\ s)\ {-}1{\rightarrow}\ (Some\ c_2,\ t) \Longrightarrow$
$(Some\ (AnnSeq\ c_0\ c_1),\ s)\ {-}1{\rightarrow}\ (Some\ (AnnSeq\ c_2\ c_1),\ t)$

AnnCond₁T: $s \in b \Longrightarrow (Some\ (AnnCond_1\ r\ b\ c_1\ c_2),\ s)\ {-}1{\rightarrow}\ (Some\ c_1,\ s)$
AnnCond₁F: $s \notin b \Longrightarrow (Some\ (AnnCond_1\ r\ b\ c_1\ c_2),\ s)\ {-}1{\rightarrow}\ (Some\ c_2,\ s)$

AnnCond₂T: $s \in b \Longrightarrow (Some\ (AnnCond_2\ r\ b\ c),\ s)\ {-}1{\rightarrow}\ (Some\ c,\ s)$
AnnCond₂F: $s \notin b \Longrightarrow (Some\ (AnnCond_2\ r\ b\ c),\ s)\ {-}1{\rightarrow}\ (None,\ s)$

AnnWhileF: $s \notin b \Longrightarrow (Some\ (AnnWhile\ r\ b\ i\ c),\ s)\ {-}1{\rightarrow}\ (None,\ s)$
AnnWhileT: $s \in b \Longrightarrow (Some\ (AnnWhile\ r\ b\ i\ c),\ s)\ {-}1{\rightarrow}$
$(Some\ (AnnSeq\ c\ (AnnWhile\ i\ b\ i\ c)),\ s)$

AnnAwait: $[\![\ s \in b;\ atom\text{-}com\ c;\ (c,\ s)\ {-}P{*}{\rightarrow}\ (Parallel\ [],\ t)\ ]\!] \Longrightarrow$
$(Some\ (AnnAwait\ r\ b\ c),\ s)\ {-}1{\rightarrow}\ (None,\ t)$

Parallel: $[\![\ i{<}length\ Ts;\ Ts!i = (Some\ c,\ q);\ (Some\ c,\ s)\ {-}1{\rightarrow}\ (r,\ t)\ ]\!]$
$\Longrightarrow (Parallel\ Ts,\ s)\ {-}P1{\rightarrow}\ (Parallel\ (Ts\ [i{:=}(r,\ q)]),\ t)$

Basic: $(Basic\ f,\ s)\ {-}P1{\rightarrow}\ (Parallel\ [],\ f\ s)$

Seq1: $All\text{-}None\ Ts \Longrightarrow (Seq\ (Parallel\ Ts)\ c,\ s)\ {-}P1{\rightarrow}\ (c,\ s)$
Seq2: $(c_0,\ s)\ {-}P1{\rightarrow}\ (c_2,\ t) \Longrightarrow (Seq\ c_0\ c_1,\ s)\ {-}P1{\rightarrow}\ (Seq\ c_2\ c_1,\ t)$

CondT: $s \in b \Longrightarrow (Cond\ b\ c_1\ c_2,\ s)\ {-}P1{\rightarrow}\ (c_1,\ s)$
CondF: $s \notin b \Longrightarrow (Cond\ b\ c_1\ c_2,\ s)\ {-}P1{\rightarrow}\ (c_2,\ s)$

WhileF: $s \notin b \Longrightarrow (While\ b\ i\ c,\ s)\ {-}P1{\rightarrow}\ (Parallel\ [],\ s)$
WhileT: $s \in b \Longrightarrow (While\ b\ i\ c,\ s)\ {-}P1{\rightarrow}\ (Seq\ c\ (While\ b\ i\ c),\ s)$

The transition rules for the similar constructs in both, annotated and non-annotated commands, are practically identical. The only difference is that the empty program is represented by *None* in the former and by *Parallel* [] in the latter.

The basic commands are executed in one step, performing the corresponding state transformation. The one-step execution of a sequential composition is determined by two rules. If the first command of the sequential composition finishes in one step, the next configuration indicates that only the second command remains to be executed. Otherwise, the first command is simply substituted by its reduction after one step.

The rules for the conditional statement lead, depending on the initial state s being an element of the set b or not, to the corresponding subprogram, leaving in both cases the state unchanged. While-statements terminate if the boolean condition is not fulfilled. Otherwise, the execution of one body followed by the original while-loop proceeds.

The transition rule *AnnAwait* formalizes the meaning of conditional atomic regions, where the body is required to be a well-formed atomic command whose execution terminates. If $s \in b$, the await-statement is executed uninterrupted in one step, provided the computation of the body terminates. If $s \notin b$, no transition is possible and the component is blocked.

Basic statements and evaluation of boolean expressions are all executed in one step. This is called a *high level* semantics, which abstracts from all the details of the evaluation of expressions.

Observe that both the preconditions denoted by r as well as the loop invariants i are merely annotations and do not play any role in the semantics. However, in the rule *AnnWhileT*, the precondition of the second *AnnWhile* has been changed to the invariant i. Although this does not influence the semantics it is important for the proof theory in §2.4.

The execution of the parallel composition of a list of annotated triples *Ts* proceeds by executing one non-*None* component of *Ts*. This form of modeling concurrency is called *interleaving*.

A terminating computation of a parallel composition of commands is a finite transition sequence starting in a state s such that in the last configuration each component program is *None*. The computation cannot be extended because there is no possible transition from *None*. For example,

$(Parallel\ [(Some\ (AnnBasic\ p\ f),\ q),\ (Some\ (AnnBasic\ p'\ g),\ q')],\ s)$
$-P1\rightarrow\ (Parallel\ [(None,\ q),\ (Some\ (AnnBasic\ p'\ g),\ q')],\ f\ s)$
$-P1\rightarrow\ (Parallel\ [(None,\ q),\ (None,\ q')],\ g\ (f\ s))$

This *one-step* semantics is particularly appropriate for concurrent languages where different executions have to be interleaved. For simplicity, we employ this style at all levels, although strictly speaking it is only necessary for component programs.

2.2.2 Definition of Semantics

There are different definitions of the semantics of programs. They differ mainly in the amount of information they provide about the input/output behavior of programs. Two of the possible definitions are the so-called *total correctness semantics* and *partial correctness semantics*. Both approaches differ in the way they deal with divergent computations. The former considers the possibility of divergence while the latter one ignores it. Other definitions [Apt and Olderog, 1991] include information about *fairness* or *deadlocks* for example. In this work we concentrate on proving partial correctness, thus, only this interpretation will be formalized.

Following Apt and Olderog, we define the *partial correctness semantics of annotated commands* by the function *ann-SEM*:

constdefs

$\textit{ann-sem} :: \alpha\ \textit{ann-com} \Rightarrow \alpha \Rightarrow \alpha\ \textit{set}$
$\textit{ann-sem}\ c \equiv \lambda s.\ \{t.\ (\textit{Some}\ c,\ s) \ {-}{*}{\rightarrow}\ (\textit{None},\ t)\}$

$\textit{ann-SEM} :: \alpha\ \textit{ann-com} \Rightarrow \alpha\ \textit{set} \Rightarrow \alpha\ \textit{set}$
$\textit{ann-SEM}\ c\ S \equiv \bigcup \textit{ann-sem}\ c\ \text{`}\ S$

The auxiliary function *ann-sem* returns for some annotated command c and some initial state s the set of all possible final states.

The semantics *ann-SEM* of an annotated command c is the union of the sets of final states that result from applying *ann-sem* c to each initial state in the set S. In other words, *ann-SEM* c S is the union of all possible final states of c executed from some state in S.

The image of a set A under a function f is denoted by f ‘ A and is predefined in the Isabelle library as $f\ \text{`}\ A \equiv \{f\,x \mid x.\ x \in A\}$.

The definition of partial correctness semantics for non-annotated programs is slightly different because of the lack of *None* as an indicator of termination:

constdefs

$\textit{sem} :: \alpha\ \textit{com} \Rightarrow \alpha \Rightarrow \alpha\ \textit{set}$
$\textit{sem}\ c \equiv \lambda s.\ \{t.\ \exists\ \textit{Ts}.\ (c,\ s)\ {-}P{*}{\rightarrow}\ (\textit{Parallel}\ \textit{Ts},\ t) \wedge \textit{All-None}\ \textit{Ts}\}$

$\textit{SEM} :: \alpha\ \textit{com} \Rightarrow \alpha\ \textit{set} \Rightarrow \alpha\ \textit{set}$
$\textit{SEM}\ c\ S \equiv \bigcup \textit{sem}\ c\ \text{`}\ S$

The semantics *SEM* satisfies several properties that we shall need in the sequel. For a property about the semantics of while-commands we define an auxiliary program called Ω .

syntax *-Omega* :: α *com* $\quad(\Omega)$

It is defined as an abbreviation of the following while-statement:

translations $\Omega \rightleftharpoons$ *While UNIV UNIV* (*Basic id*)

where *UNIV* stands for the universal set of a fixed type, i.e. $\{x.\ True\}$ and *id* represents the identity transformation. This particular program enjoys the following property:

lemma *SEM-Omega*: *SEM* Ω $S = \{\}$

The primitive recursive function *fwhile* defines a sequence of deterministic programs that simulates the behavior of a while-statement:

consts *fwhile* :: α *bexp* $\Rightarrow$ α *com* $\Rightarrow$ *nat* $\Rightarrow$ α *com*
primrec
  fwhile b c $0 = \Omega$
  fwhile b c (*Suc* n) = *Cond* b (*Seq* c (*fwhile* b c n)) (*Basic id*)

We prove the following lemmas about *SEM* as stated in [Apt and Olderog, 1991] (the proofs in the book are left as an exercise). We briefly review the main steps of our mechanized version.:

lemma *SEM-mono*: $X \subseteq Y \Longrightarrow$ *SEM* c $X \subseteq$ *SEM* c Y

The proof is automatic.

lemma *SEM-Seq*: *SEM* (*Seq* c_1 c_2) X = *SEM* c_2 (*SEM* c_1 X)

Proof. The $\subseteq$-inclusion is proved using the following lemma:

lemma *SEM-Seq-onlyif*:
  $[\![$ (*Seq* c_1 c_2, s) $-P{*}\rightarrow$ (*Parallel Ts*, t); *All-None Ts* $]\!]$
  $\Longrightarrow \exists\, y$ *Rs*. (c_1, s) $-P{*}\rightarrow$ (*Parallel Rs*, y) $\wedge$ *All-None Rs* $\wedge$
    (c_2, y) $-P{*}\rightarrow$ (*Parallel Ts*, t)

The proof relies on the following auxiliary lemma solved by induction on n:

lemma *SEM-Seq-onlyif-aux*:
$[\![$ $(Seq\ c_1\ \ c_2,\ s)\ {-Pn}{\rightarrow}\ (Parallel\ Ts,\ t)$; $All\text{-}None\ Ts$ $]\!]$
$\Longrightarrow \exists\, y\ m\ Rs.\ (c_1,\ s)\ {-P{*}}{\rightarrow}\ (Parallel\ Rs,\ y) \wedge All\text{-}None\ Rs\ \wedge$
$(c_2,\ y)\ {-Pm}{\rightarrow}\ (Parallel\ Ts,\ t) \wedge m \le n$

For the $\supseteq$-inclusion we use the lemma:

lemma *SEM-Seq-if*:
$[\![$ $(c_1,\ s_1)\ {-P{*}}{\rightarrow}\ (Parallel\ Ts,\ s_2)$; $All\text{-}None\ Ts$;
$(c_2,\ s_2)\ {-P{*}}{\rightarrow}\ (Parallel\ Rs,\ s_3)$; $All\text{-}None\ Rs$ $]\!]$
$\Longrightarrow (Seq\ c_1\ c_2,\ s_1)\ {-P{*}}{\rightarrow}\ (Parallel\ Rs,\ s_3)$

which is proven by induction on the length of the transition sequence given by $(c_1,\ s_1)\ {-P{*}}{\rightarrow}\ (Parallel\ Ts,\ s_2)$. □

lemma *SEM-Seq-assoc*:
$SEM\ (Seq\ (Seq\ c_1\ c_2)\ c_3)\ X = SEM\ (Seq\ c_1\ (Seq\ c_2\ c_3))\ X$

The proof is trivial.

lemma *SEM-Cond*:
$SEM\ (Cond\ b\ c_1\ c_2)\ X = SEM\ c_1\ (X \cap b) \cup SEM\ c_2\ (X \cap -b)$

where $-b$ represents the complement set of b. Both inclusions are easily solved by properly manipulating the transitive closure and doing case analysis on the conditional statement.

lemma *SEM-While*: $SEM\ (While\ b\ i\ c) = (\lambda x.\ \bigcup k.\ SEM\ (fwhile\ b\ c\ k)\ x)$

Proof. The $\subseteq$-inclusion is proved using the lemma:

lemma *SEM-While-onlyif*:
$[\![$ $(While\ b\ i\ c,\ s)\ {-Pn}{\rightarrow}\ (Parallel\ Ts,\ t)$; $All\text{-}None\ Ts$ $]\!]$
$\Longrightarrow \exists\, k.\ (fwhile\ b\ c\ k,\ s)\ {-P{*}}{\rightarrow}\ (Parallel\ Ts,\ t)$

which is proved by complete induction on n. With the induction hypothesis we obtain the subgoal:

$[\![$ $\forall\, m < n.\ (\forall\, s.\ (While\ b\ i\ c,\ s)\ {-Pm}{\rightarrow}\ (Parallel\ Ts,\ t) \wedge All\text{-}None\ Ts$
$\longrightarrow (\exists\, k.\ (fwhile\ b\ c\ k,\ s)\ {-P{*}}{\rightarrow}\ (Parallel\ Ts,\ t)))$;

$(\mathit{While}\ b\ i\ c,\ s)\ -Pn\rightarrow\ (\mathit{Parallel}\ Ts,\ t);\ \mathit{All\text{-}None}\ Ts\ ]\!]$
$\Longrightarrow \exists\, k.\ (\mathit{fwhile}\ b\ c\ k,\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$

We want to find an appropriate k satisfying the conclusion. If $s \notin b$, it suffices to take $k = 1$. If $s \in b$, we obtain by case analysis:

$$(\mathit{Seq}\ c\ (\mathit{While}\ b\ i\ c),\ s)\ -Pm\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$$

where $n = \mathit{Suc}\ m$. By *SEM-Seq-onlyif-aux* we can split the computation of the sequential composition into the computations of its two components. Then, for some y we obtain:

$$(c,\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Rs,\ y) \wedge \mathit{All\text{-}None}\ Rs$$
$$(\mathit{While}\ b\ i\ c,\ y)\ -Pm'\rightarrow\ (\mathit{Parallel}\ Ts,\ t) \wedge m' \le m$$

we apply the induction hypothesis obtaining for some k',

$$(\mathit{fwhile}\ b\ c\ k',\ y)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$$

choosing $k = k' + 1$ the conclusion becomes

$$(\mathit{Cond}\ b\ (\mathit{Seq}\ c\ (\mathit{fwhile}\ b\ c\ k))\ \mathit{Basic}\ id,\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$$

but $s \in b$ holds, so it can be simplified to

$$(\mathit{Seq}\ c\ \ (\mathit{fwhile}\ b\ c\ k),\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$$

which we prove via *SEM-Seq-if*.

The $\supseteq$-inclusion requires also an auxiliary lemma:

lemma *SEM-While-if*:
$[\![\ (\mathit{fwhile}\ b\ c\ k,\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t);\ \mathit{All\text{-}None}\ Ts\ ]\!]$
$\Longrightarrow (\mathit{While}\ b\ i\ c,\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$

proved by induction on k as follows. The base case is easy; since Ω does not terminate, there is a contradiction among the premises.

For the induction step, we first distinguish whether the transitive closure in the premise performs at least one step. If not, the proof is trivial. Otherwise, from the definition of *fwhile* we obtain a conditional statement, thus by case analysis there are two possible situations:

1. If $s \in b$, then $(\mathit{Seq}\ c\ \ (\mathit{fwhile}\ b\ c\ n),\ s)\ -P{*}\rightarrow\ (\mathit{Parallel}\ Ts,\ t)$. Decomposing the computation of the sequential composition by applying

the lemma *SEM-Seq-onlyif-aux* and using the induction hypothesis we obtain for some Rs and some state y

$$(c,\ s)\ {-P{*}{\rightarrow}}\ (\mathit{Parallel}\ Rs,\ y) \wedge \mathit{All\text{-}None}\ Rs \wedge$$
$$(\mathit{While}\ b\ i\ c,\ y)\ {-P{*}{\rightarrow}}\ (\mathit{Parallel}\ Ts,\ t)$$

as premises. Since we have $s \in b$, the conclusion becomes

$$(\mathit{Seq}\ c\ (\mathit{While}\ b\ i\ c),\ s)\ {-P{*}{\rightarrow}}\ (\mathit{Parallel}\ Ts,\ t)$$

The proof follows by applying the lemma *SEM-Seq-if*.

2. If $s \notin b$, then the conclusion becomes

$$(\mathit{Parallel}\ [],\ s)\ {-P{*}{\rightarrow}}\ (\mathit{Parallel}\ Ts,\ t)$$

We need to show that $s = t$ and $Ts = []$. This follows easily from the premise $(\mathit{Basic}\ id,\ s)\ {-P{*}{\rightarrow}}\ (\mathit{Parallel}\ Ts,\ t)$ and the lemma

$(\mathit{Parallel}\ [],\ s)\ {-Pn{\rightarrow}}\ (\mathit{Parallel}\ Ts,\ t) \Longrightarrow Ts = [] \wedge n = 0 \wedge s = t$

□

2.3 Validity of Correctness Formulas

Before introducing the proof system that shows us the rules to derive correct programs, we need to formalize what we mean by "correct", and in particular correct in the sense of partial correctness. Informally, we say that a program is correct if it satisfies the intended input/output relation, where input and output are usually described as predicates over states, or equivalently as sets of states. Thus, a program specification consists of a triple, also called *Hoare triple*, of the form $\{p\}\ c\ \{q\}$ where c is a program and p and q are the corresponding *precondition* and *postcondition*. The precondition describes the set of initial or input states, in which the program c is started, and the postcondition describes the set of final or output states.

Then, we say that a formula $\{p\}\ c\ \{q\}$ is *valid* (or true) in the sense of partial correctness iff every terminating computation of c that starts in a state s satisfying p ends in a state satisfying q. This definition does not take diverging computations of c into account. Following [Apt and Olderog, 1991], we formalize this interpretation as set theoretic inclusions.

For a command c of type α *com* we define *validity of a partial correctness formula*, and write $\Vdash p\ c\ q$ as follows:

constdefs
  com-validity $:: \alpha\ assn \Rightarrow \alpha\ com \Rightarrow \alpha\ assn \Rightarrow bool \qquad (\Vdash\ \text{- - -})$
 $\Vdash p\ c\ q \equiv SEM\ c\ p \subseteq q$

where p and q are sets of initial and final states, respectively.

Validity of a partial correctness formula for an annotated command is defined analogously:

constdefs
  ann-com-validity $:: \alpha\ ann\text{-}com \Rightarrow \alpha\ assn \Rightarrow bool \qquad (\models\ \text{- -})$
 $\models c\ q \equiv ann\text{-}SEM\ c\ (pre\ c) \subseteq q$

The only difference with programs of type α *com* is that we do not need an extra argument for the precondition since it is included as part of the annotated command and can be extracted via the function *pre*.

2.4 The Proof System

Given a correctness formula, we can reason about its validity directly in terms of the semantics. This methodology is commonly known as *operational* or *behavioral* reasoning and basically consists on observing the effects of the computation by unfolding the steps according to the rules of the operational semantics. This procedure should be repeated for each possible initial state. Obviously, this is very tedious for non-trivial programs (or even impossible since the set of initial states may be infinite). Programmers using this procedure tend to give an informal account of the possible behaviors of the program for certain inputs, but experience has shown that this lack of structure is bound to fail due to a non-exhaustive study of the possibilities.

Hoare's approach to program verification is based on proving that a given correctness formula is derivable in a system of axioms and inference rules which are syntax oriented. The relation between the program syntax and its semantics is studied exclusively in the proof of soundness of the system. Once this is achieved, we can forget the semantics completely and just use the system of rules for program verification.

The main properties of such systems are soundness and completeness. The rules of a *sound* system inductively define a set of correct programs. If the system is also *complete* then it defines the set of *all* correct programs.

We define three systems of rules, one for each level of the language. However, due to the interdependencies among the different levels, the three sets which are inductively defined from the three systems of rules are mutually recursive. Thus, they must be declared in a single inductive set definition. Atomic and parallel programs share the rules for the common constructors.

In this section we progressively show the three systems and give the full set of rules only at the end. We start with the system for atomic programs. The definition of the rules and the soundness proof is independent of the other rules. We continue with the system for component programs which depends on the previous one and finish with the rule for parallel composition.

2.4.1 Proof System for Atomic Programs

The set of derivable correctness formulas of atomic programs is inductively defined by the set:

consts $oghoare :: (\alpha\ assn \times \alpha\ com \times \alpha\ assn)\ set$

The syntax $\Vdash\ p\ c\ q$ denotes that the triple $(p,\ c,\ q)$ is an element of the inductively defined set. This amounts to say that it can be derived in the system.

syntax $\text{-}oghoare :: \alpha\ assn \Rightarrow \alpha\ com \Rightarrow \alpha\ assn \Rightarrow bool \qquad (\Vdash\ \text{- - -})$
translations $\Vdash\ p\ c\ q \rightleftharpoons (p,\ c,\ q) \in oghoare$

A complete system contains at least one axiom or inference rule for each constructor of the language. The rules defining this set are shown in table 2.1.

2.4.2 Proof System for Component Programs

Correctness formulas for component programs are not just annotated with a precondition and a postcondition. As explained in §2.1 they appear as proof outlines, where each command is preceded by its precondition. The set of derivable proof outlines is defined by the constant:

consts $ann\text{-}hoare :: (\alpha\ ann\text{-}com \times \alpha\ assn)\ set$

A pair $(c,\ q)$ is an element of the set if it has derivation in the system, denoted $\vdash\ c\ q$:

Basic:	$\Vdash \{s.\ f\ s \in q\}\ (Basic\ f)\ q$
Seq:	$[\![\Vdash p\ c_1\ r; \Vdash r\ c_2\ q\ ]\!] \Longrightarrow \Vdash p\ (Seq\ c_1\ c_2)\ q$
Cond:	$[\![\Vdash (p \cap b)\ c_1\ q; \Vdash (p \cap -b)\ c_2\ q\ ]\!]$ $\Longrightarrow \Vdash p\ (Cond\ b\ c_1\ c_2)\ q$
While:	$[\![\Vdash (p \cap b)\ c\ p\ ]\!] \Longrightarrow \Vdash p\ (While\ b\ i\ c)\ (p \cap -b)$
Conseq:	$[\![\ p' \subseteq p; \Vdash p\ c\ q\ ;\ q \subseteq q'\ ]\!] \Longrightarrow \Vdash p'\ c\ q'$

Table 2.1: Proof rules for atomic commands.

syntax *-ann-hoare* :: α *ann-com* $\Rightarrow$ α *assn* $\Rightarrow$ *bool* ($\vdash$ - -)
translations $\vdash c\ q \rightleftharpoons (c,\ q) \in$ *ann-hoare*

The formation rules for proof outlines are shown in table 2.2. They look unusual because preconditions are hidden as part of the commands' syntax. The consequence rule does not permit to strengthen the precondition. However, this possibility appears embedded in each of the other rules. The following result shows that this system is equivalent to the standard presentation.

Equivalence of Proof Systems.

Let $\vdash_{st} p\ \kappa\ q$ stand for provability of the correctness formula $p\ \kappa\ q$ in some standard system. By κ we mean commands without any annotation other than loop invariants. The relation $c \sim \kappa$ means that both commands are equal except for the annotations (loop invariants must also be equal). Then,

1. $\vdash_{st} p\ \kappa\ q \implies \exists c.\ \vdash c\ q\ \wedge\ p \subseteq pre\ c\ \wedge\ \kappa \sim c$
2. $\vdash c\ q \implies \exists \kappa.\ \vdash_{st} (pre\ c)\ \kappa\ q\ \wedge\ \kappa \sim c$

Proof. Both directions are proven by rule induction on $\vdash_{st}$ and $\vdash$ respectively. We have not formalized them in Isabelle, but we hope to convince the reader that the system above can be used without any loss of generality:

1. To prove the first implication we have to consider six cases, one for each rule (the standard system contains only one rule for the conditional-statement). We only show two representative ones:

AnnBasic:	$r \subseteq \{s.\ f\ s \in q\} \Longrightarrow \vdash (\mathit{AnnBasic}\ r\ f)\ q$
AnnSeq:	$[\![\vdash c_0\ \mathit{pre}\ c_1; \vdash c_1\ q\ ]\!] \Longrightarrow \vdash (\mathit{AnnSeq}\ c_0\ c_1)\ q$
AnnCond$_1$:	$[\![\ r \cap b \subseteq \mathit{pre}\ c_1; \vdash c_1\ q;\ r \cap -b \subseteq \mathit{pre}\ c_2; \vdash c_2\ q\ ]\!] \Longrightarrow$ $\vdash (\mathit{AnnCond}_1\ r\ b\ c_1\ c_2)\ q$
AnnCond$_2$:	$[\![\ r \cap b \subseteq \mathit{pre}\ c; \vdash c\ q;\ r \cap -b \subseteq q\ ]\!] \Longrightarrow$ $\vdash (\mathit{AnnCond}_2\ r\ b\ c)\ q$
AnnWhile:	$[\![\ r \subseteq i;\ i \cap b \subseteq \mathit{pre}\ c; \vdash c\ i;\ i \cap -b \subseteq q\ ]\!]$ $\Longrightarrow \vdash (\mathit{AnnWhile}\ r\ b\ i\ c)\ q$
AnnAwait:	$[\![\ \mathit{atom\text{-}com}\ c; \Vdash (r \cap b)\ c\ q\ ]\!] \Longrightarrow \vdash (\mathit{AnnAwait}\ r\ b\ c)\ q$
AnnConseq:	$[\![\vdash c\ q;\ q \subseteq q'\ ]\!] \Longrightarrow \vdash c\ q'$

Table 2.2: Proof rules for annotated commands.

Basic Suppose that the equivalent non-annotated basic command is called Basic in some standard system. We have to prove

$$\vdash_{st} \{s.\ f\ s \in q\}\ (\mathsf{Basic}\ f)\ q \implies$$
$$\exists c.\ \vdash c\ q\ \wedge\ \{s.\ f\ s \in q\} \subseteq \mathit{pre}\ c\ \wedge\ (\mathsf{Basic}\ f) \sim c$$

The annotated command *AnnBasic* $\{s.\ f\ s \in q\}\ f$ fulfills the requirements.

Seq Suppose $\vdash_{st} p\ (c_1; c_2)\ q$. From the rule of sequential composition we have for some r, $\vdash_{st}\ p\ c_1\ r$ and $\vdash_{st}\ r\ c_2\ q$. By induction hypothesis we obtain for some *ca* and some *cb* the assumptions:

$$\vdash \mathit{ca}\ r\ \wedge\ p \subseteq \mathit{pre}\ \mathit{ca}\ \wedge\ c_1 \sim \mathit{ca}, \text{ and}$$
$$\vdash \mathit{cb}\ q\ \wedge\ r \subseteq \mathit{pre}\ \mathit{cb}\ \wedge\ c_2 \sim \mathit{cb}.$$

We want to find a command c such that

$$\vdash c\ q\ \wedge\ p \subseteq \mathit{pre}\ c\ \wedge\ c_1; c_2 \sim c$$

Taking $c = \mathit{AnnSeq}\ \mathit{ca}\ \mathit{cb}$ we have from $c_1 \sim \mathit{ca}$ and $c_2 \sim \mathit{cb}$ that $c_1; c_2 \sim \mathit{AnnSeq}\ \mathit{ca}\ \mathit{cb}$.

From $p \subseteq \mathit{pre}\ \mathit{ca}$ and $\mathit{pre}\ \mathit{AnnSeq}\ \mathit{ca}\ \mathit{cb} = \mathit{pre}\ \mathit{ca}$ (by definition of pre), we obtain $p \subseteq \mathit{pre}\ \mathit{AnnSeq}\ \mathit{ca}\ \mathit{cb}$. By the rule of consequence,

$$\vdash \mathit{ca}\ r\ \wedge\ r \subseteq \mathit{pre}\ \mathit{cb} \Longrightarrow \vdash \mathit{ca}\ \mathit{pre}\ \mathit{cb}$$

And finally, by the rule for sequential composition,

$$\vdash \mathit{ca}\ (\mathit{pre}\ \mathit{cb}) \wedge\ \vdash \mathit{cb}\ q \Longrightarrow \vdash \mathit{AnnSeq}\ \mathit{ca}\ \mathit{cb}\ q$$

2. We illustrate the opposite direction with two cases not considered before:

AnnAwait Suppose the await-command in the standard system is called Await. We want to prove
$\vdash (\mathit{AnnAwait}\ r\ b\ c)\ q \Longrightarrow\ \exists\kappa. \vdash_{st} r\ \kappa\ q\ \wedge \kappa \sim (\mathit{AnnAwait}\ r\ b\ c)$
By the proof rule *AnnAwait*, we know that $\Vdash (r \cap b)\ c\ q$. The system of rules for atomic programs is like the standard system and since c is an atomic command it does not contain assertions, thus we can assume $\vdash_{st} (r \cap b)\ c\ q$.
If we choose $\kappa = \mathsf{Await}\ b\ c$ we can derive $\vdash_{st}\ r\ (\mathsf{Await}\ b\ c)\ q$ and obviously $\mathit{AnnAwait}\ r\ b\ c \sim \mathsf{Await}\ b\ c$ holds.

AnnConseq Suppose $\vdash c'\ q'$. By the rule of consequence we have the premises $\vdash c'\ q$ and $q \subseteq q'$ for some q. By hypothesis there is a non-annotated command κ such that $\vdash_{st} \mathit{pre}\ c'\ \kappa\ q$ and $\kappa \sim c'$. Obviously, κ is the searched program:

$$\vdash_{st} \mathit{pre}\ c'\ \kappa\ q\ \wedge\ q \subseteq q' \Longrightarrow \vdash_{st} \mathit{pre}\ c'\ \kappa\ q'$$

This concludes the proof of equivalence of our system of rules and a standard one like the one in [Apt and Olderog, 1991], where preconditions are not directly attached to commands. □

2.4.3 Proof System for Parallel Programs

Proof of correctness of parallel programs is much more demanding than the sequential case. The problem is that different components can interfere with each other via shared variables. Unfortunately, proving that all proof outlines are correct independently of the environment is not sufficient to conclude that the input/output specification of a parallel composition is the intersection (assertions are modeled as sets) of the input/output specification of each component. We also need to guarantee that the proof outline of any component is not falsified by the execution of the others. This property, called *interference freedom of proof outlines*, is determined by the predicate *interfree*. Its definition requires a number of auxiliary concepts:

- An assertion p is invariant under execution of an atomic command a iff $\models (p \cap pre\ a)\ a\ p$.
- An atomic command a *does not interfere* with a standard proof outline $c\ q$ iff the following two conditions hold:
    1. $\models (q \cap pre\ a)\ a\ q$,
    2. For any assertion p within c: $\models (p \cap pre\ a)\ a\ p$
- Standard proof outlines $c_1\ q_1, \ldots, c_n\ q_n$ are interference free if no assignment or atomic region of a program c_i interferes with the proof outline $c_j\ q_j$ of another component with $i \neq j$.

Given two component program's proof outlines $c\ q$ and $c'\ q'$, showing interference freedom means proving that all assertions in the former remain invariant under execution of all assignments or atomic regions in the latter, and vice versa.

Atomic commands are collected by the function *atomics* which, given an annotated command, returns the set of all pairs $(r,\ a)$ where a is either the body of an *AnnAwait*-command, or a *Basic*-command (from an *AnnBasic*-command) and r is the corresponding precondition.

consts $atomics :: \alpha\ ann\text{-}com \Rightarrow (\alpha\ assn \times \alpha\ com)\ set$
primrec
$atomics\ (AnnBasic\ r\ f) = \{(r,\ Basic\ f)\}$
$atomics\ (AnnSeq\ c_1\ c_2) = atomics\ c_1 \cup atomics\ c_2$
$atomics\ (AnnCond_1\ r\ b\ c_1\ c_2) = atomics\ c_1 \cup atomics\ c_2$
$atomics\ (AnnCond_2\ r\ b\ c) = atomics\ c$
$atomics\ (AnnWhile\ r\ b\ i\ c) = atomics\ c$
$atomics\ (AnnAwait\ r\ b\ c) = \{(r \cap b,\ c)\}$

The set of all assertions of an annotated command (including loop invariants) is collected by the function *assertions*:

consts $assertions :: \alpha\ ann\text{-}com \Rightarrow (\alpha\ assn)\ set$
primrec
$assertions\ (AnnBasic\ r\ f) = \{r\}$
$assertions\ (AnnSeq\ c_1\ c_2) = assertions\ c_1 \cup assertions\ c_2$

Parallel:

$$\llbracket\ \forall\, i < \mathit{length}\ \mathit{Ts}.\ \exists\, c\ q.\ \mathit{Ts}!i = (\mathit{Some}\ c,\ q) \wedge \vdash c\ q;\ \mathit{interfree}\ \mathit{Ts}\ \rrbracket \Longrightarrow$$
$$\Vdash (\bigcap i \in \{i.\ i < \mathit{length}\ \mathit{Ts}\}.\ \mathit{pre}\ (\mathit{the}\ (\mathit{com}\ (\mathit{Ts}!i))))$$
$$\mathit{Parallel}\ \mathit{Ts}$$
$$(\bigcap i \in \{i.\ i < \mathit{length}\ \mathit{Ts}\}.\ \mathit{post}\ (\mathit{Ts}!i))$$

Table 2.3: Proof rule for parallel programs.

$\mathit{assertions}\ (\mathit{AnnCond}_1\ r\ b\ c_1\ c_2) = \{r\} \cup \mathit{assertions}\ c_1 \cup \mathit{assertions}\ c_2$
$\mathit{assertions}\ (\mathit{AnnCond}_2\ r\ b\ c) = \{r\} \cup \mathit{assertions}\ c$
$\mathit{assertions}\ (\mathit{AnnWhile}\ r\ b\ i\ c) = \{r,\ i\} \cup \mathit{assertions}\ c$
$\mathit{assertions}\ (\mathit{AnnAwait}\ r\ b\ c) = \{r\}$

The interference freedom test in one direction, i.e. where the assertions in $(co,\ q)$ are checked for invariance against the atomic actions in co', is realized by the function *interfree-aux*:

constdefs $\mathit{interfree\text{-}aux} :: (\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha\ \mathit{assn} \times \alpha\ \mathit{ann\text{-}com\text{-}op}) \Rightarrow \mathit{bool}$
$\mathit{interfree\text{-}aux} \equiv \lambda(co,\ q,\ co').\ co' = \mathit{None} \vee$
$\quad (\forall\, (r,\ a) \in \mathit{atomics}\ (\mathit{the}\ co').\ \Vdash (q \cap r)\ a\ q\ \wedge$
$\quad (co = \mathit{None} \vee (\forall\, p \in \mathit{assertions}\ (\mathit{the}\ co).\ \Vdash (p \cap r)\ a\ p)))$

The function *interfree-aux* must be applied to all possible combinations of component programs, except for a component program with itself. Hence, the definition of *interfree* becomes:

constdefs $\mathit{interfree} :: (\alpha\ \mathit{ann\text{-}triple\text{-}op})\ \mathit{list} \Rightarrow \mathit{bool}$
$\mathit{interfree}\ \mathit{Ts} \equiv \forall\, i\ j.\ i < \mathit{length}\ \mathit{Ts} \wedge j < \mathit{length}\ \mathit{Ts} \wedge i \neq j \longrightarrow$
$\quad \mathit{interfree\text{-}aux}\ (\mathit{com}\ (\mathit{Ts}!i),\ \mathit{post}\ (\mathit{Ts}!i),\ \mathit{com}\ (\mathit{Ts}!j))$

The rule for parallel composition shown in table 2.3 claims that if all component programs are correct and interference free, then the parallel composition satisfies the formula where the precondition is the intersection of all the components' preconditions and the postcondition is the intersection of all the components' postconditions. Each element of *Ts* is a pair of an optional command α *ann-com-op* and a postcondition α *assn*. The function *post* extracts the postcondition, *com* extracts the optional command α *ann-com-op*, the predefined function *the* extracts the command c from *Some* c (by assumption all commands are wrapped up in *Some*), and finally *pre* extracts the precondition. This rule together with the rules for atomic programs constitute the system of rules for parallel programs.

Because of the interdependencies among the rules, the full system consisting of the rules that define the set *oghoare* and the rules that define the set *ann-hoare* must be declared simultaneously in a so-called *mutually inductive definition*:

inductive *oghoare* *ann-hoare*
intros

$AnnBasic$: $r \subseteq \{s.\ f\ s \in q\} \Longrightarrow \vdash (AnnBasic\ r\ f)\ q$

$AnnSeq$: $[\![\vdash c_0\ pre\ c_1; \vdash c_1\ q\ ]\!] \Longrightarrow \vdash (AnnSeq\ c_0\ c_1)\ q$

$AnnCond_1$: $[\![r \cap b \subseteq pre\ c_1; \vdash c_1\ q;\ r \cap -b \subseteq pre\ c_2; \vdash c_2\ q\ ]\!]$
$\Longrightarrow \vdash (AnnCond_1\ r\ b\ c_1\ c_2)\ q$

$AnnCond_2$: $[\![r \cap b \subseteq pre\ c; \vdash c\ q;\ r \cap -b \subseteq q\ ]\!] \Longrightarrow \vdash (AnnCond_2\ r\ b\ c)\ q$

$AnnWhile$: $[\![r \subseteq i;\ i \cap b \subseteq pre\ c; \vdash c\ i;\ i \cap -b \subseteq q\ ]\!]$
$\Longrightarrow \vdash (AnnWhile\ r\ b\ i\ c)\ q$

$AnnAwait$: $[\![atom\text{-}com\ c; \Vdash (r \cap b)\ c\ q\ ]\!] \Longrightarrow \vdash (AnnAwait\ r\ b\ c)\ q$

$AnnConseq$: $[\![\vdash c\ q;\ q \subseteq q'\ ]\!] \Longrightarrow \vdash c\ q'$

$Parallel$: $[\![\forall i < length\ Ts.\ \exists c\ q.\ Ts!i = (Some\ c,\ q) \wedge \vdash c\ q;\ interfree\ Ts\ ]\!]$
$\Longrightarrow \Vdash (\bigcap i \in \{i.\ i < length\ Ts\}.\ pre\ (the\ (com\ (Ts!i))))$
$Parallel\ Ts$
$(\bigcap i \in \{i.\ i < length\ Ts\}.\ post\ (Ts!i))$

$Basic$: $\Vdash \{s.\ f\ s \in q\}\ (Basic\ f)\ q$

Seq: $[\![\Vdash p\ c_1\ r; \Vdash r\ c_2\ q\ ]\!] \Longrightarrow \Vdash p\ (Seq\ c_1\ c_2)\ q$

$Cond$: $[\![\Vdash (p \cap b)\ c_1\ q; \Vdash (p \cap -b)\ c_2\ q\ ]\!] \Longrightarrow \Vdash p\ (Cond\ b\ c_1\ c_2)\ q$

$While$: $[\![\Vdash (p \cap b)\ c\ p\ ]\!] \Longrightarrow \Vdash p\ (While\ b\ i\ c)\ (p \cap -b)$

$Conseq$: $[\![p' \subseteq p; \Vdash p\ c\ q\ ;\ q \subseteq q'\ ]\!] \Longrightarrow \Vdash p'\ c\ q'$

Like in the definition of the semantics, atomic and parallel programs share the rules for the common constructors. Thus, to denote that a triple (p, c, q), where c is a parallel program, is derivable in the system we use the same syntax as for atomic programs $\Vdash\ p\ c\ q$.

We can refer to a particular rule, for example *Seq*, by writing the prefix *oghoare-ann-hoare*, i.e. *oghoare-ann-hoare.Seq*.

2.4.4 Auxiliary Variables

An important aspect of the Owicki-Gries method is the use of auxiliary variables. They augment the program with additional information for proof purposes. Therefore, auxiliary variables should neither affect the control flow nor the data flow of the program. In fact, they are only allowed to appear in assignments of the form $a := t$, where a is an auxiliary variable. Since auxiliary variables may not appear in boolean expressions or in assignments to program variables, they are superfluous to the real computation and can therefore be eliminated.

Auxiliary variables record information about the course of the computation in a program which cannot in general be captured by the program variables alone. There are two main kinds of auxiliary variables:

- *History variables:* only one such auxiliary variable is required for the full parallel program. It records the values of all program variables atomically with every assignment or atomic region. At the end of the computation the history variable contains the full sequence of states that the execution has gone through.

- *Location variables:* in this case, a different auxiliary variable of this kind is introduced for each component of a parallel program. These variables keep track of the location where control flow resides at each moment of the computation by means of labels, where a different label is needed for every possible control point.

There are several well-known systematic procedures for the introduction of auxiliary variables of both kinds [Owicki, 1975, Apt and Olderog, 1991, Best, 1996, de Roever *et al.*, 2000]. Such general procedures are essential for the completeness proof of the Owicki-Gries method because they work for every possible program. However, history variables are too complicated in practice and the general procedure for location variables introduces too many auxiliary variables. A more clever proof for a particular program can normally extract the needed information from a few suitable auxiliary variables.

The general approach consists of extending the program by the assignments to auxiliary variables, proving the correctness of the extended program and then deleting the added assignments. This last step is done with the elimination rule

$$p \ c \ q \Longrightarrow p \ c^* \ q$$

where for some set of auxiliary variables A used in the program c such that $(\textit{free variables in } q) \cap A = \emptyset$, the program c^* is obtained from c by deleting all assignments to the variables in A.

The need for the auxiliary variable rule is a recurrent issue in research about verification systems for parallel programs. Stirling describes the need for this elimination rule as a “major disadvantage” [Stirling, 1988]. Further studies have demonstrated that it is possible to design a complete verification calculus for parallel programs with shared variables where the auxiliary structure is only a part of the logic, so that the program text need not be modified [Soundararajan, 1984, Stølen, 1991].

Our current proof system is incomplete because there is no rule for removing auxiliary variables. We can prove the correctness of the extended program, but it is left to the user to ensure that auxiliary variables are used correctly.

2.5 Soundness

We are interested in the following soundness property:

$$\Vdash p \ c \ q \Longrightarrow \Vdash\!= p \ c \ q$$

that whenever a correctness formula for a parallel program is derivable in the proof system then it is also valid in the sense of partial correctness.

Properties of inductively defined sets are usually proven by rule induction. The theorem describing this proof principle is automatically generated by Isabelle for every inductively defined set. The idea is to prove that a given property is true for all axioms of the system and that it is preserved by all inference rules. Since an inductively defined set is the least set closed under the given axioms and rules, every element of the set that has a derivation in the system satisfies the property.

The proof of soundness proceeds in three stages that correspond to the three subsystems. In §2.5.1, we prove soundness of the subsystem for atomic programs. This result is necessary for the soundness proof of the subsystem for component (annotated) programs in §2.5.2. Section §2.5.3 presents the soundness of the rule for parallel programs. With these results, we finally prove soundness of the full system.

2.5.1 Soundness of the System for Atomic Programs

The proof is done by rule induction on the set of rules defining *oghoare*.

theorem *atom-hoare-sound*: $[\![\Vdash p\ c\ q;\ atom\text{-}com\ c\ ]\!] \Longrightarrow \models p\ c\ q$

This amounts to proving the soundness of each rule separately. We require that the program be atomic, consequently the subgoal concerning the *Parallel* rule is trivially eliminated. The proofs for the other rules follow directly from the lemmas about the semantics *SEM* (cf. §2.3).

2.5.2 Soundness of the System for Component Programs

A correctness formula is valid in the sense of partial correctness iff whenever a program c started in a state satisfying the precondition terminates, then the final state satisfies the postcondition. Observe that this definition does not mention the intermediate assertions. This is fine for non-annotated programs, but in our case satisfiability of the intermediate annotations is also relevant.

Informally speaking, proof outlines fulfill the property that whenever the control of c in a given computation starting in a state $s \in p$ reaches a point annotated by an assertion, this assertion is true. This property is called *strong soundness*. The standard soundness property follows trivially from this theorem.

Strong Soundness for Component Programs

This property of proof outlines is formally proven in the following theorem:

theorem *Strong-Soundness*:
$[\![(Some\ c,\ s) -{*}{\rightarrow} (co,\ t);\ s \in pre\ c;\ \vdash c\ q\ ]\!]$
$\Longrightarrow if\ co = None\ then\ t \in q\ else\ t \in pre\ (the\ co)$

where *the* is a predefined function that extracts c from *Some c*.

Proof. The proof is by induction on the length of the computation. However, the *Strong-Soundness* theorem is not yet suitably formulated. To be able to apply the induction hypothesis we need the fact that the program rest, i.e. *the co* is also derivable in the system:

lemma *Strong-Soundness-aux*: $[\![\ (Some\ c,\ s)\ -{*}{\rightarrow}\ (co,\ t);\ s \in pre\ c;\ \vdash\ c\ q\ ]\!]$
$\Longrightarrow if\ co = None\ then\ t \in q\ else\ t \in pre\ (the\ co) \wedge \vdash (the\ co)\ q$

If the length of the computation is 0 then $co = (Some\ c)$ and $t = s$. By hypothesis we know that $s \in pre\ c$, then $t \in pre\ c$. Suppose the length is now positive. Then, for some co' and t' we have

$$(Some\ c,\ s)\ -{*}{\rightarrow}\ (co',\ t')\ -1{\rightarrow}\ (co,\ t)$$

co' cannot be *None* because there is no possible transition from *None* in the system *ann-transition*. Thus, there is a c' so that $co' = Some\ c'$. By the induction hypothesis we know that $t' \in pre\ c'$ and $\vdash\ c'\ q$. The proof follows by rule induction on the last step. This is achieved via the following auxiliary lemma:

lemma *Strong-Soundness-aux-aux*:
$[\![\ (co,\ s)\ -1{\rightarrow}\ (co',\ t);\ co = Some\ c;\ s \in pre\ c;\ \vdash\ c\ q\ ]\!]$
$\Longrightarrow if\ co' = None\ then\ t \in q\ else\ t \in pre\ (the\ co') \wedge \vdash (the\ co')\ q$

We discuss three representative cases from the ten that result from applying rule induction on $(co,\ s)\ -1{\rightarrow}\ (co',\ t)$.

Seq2: From this rule we obtain $(Some\ c_0,\ s)\ -1{\rightarrow}\ (Some\ c_2,\ t)$ in the premises. After applying the induction hypothesis and using $pre\ (AnnSeq\ c_0\ c_1) = pre\ c_0$ we obtain:

$[\![\ (Some\ c_0,\ s)\ -1{\rightarrow}\ (Some\ c_2,\ t);\ s \in pre\ c_0;\ \vdash (AnnSeq\ c_0\ c_1)\ q;$
$\forall q.\ \vdash\ c_0\ q \longrightarrow t \in pre\ c_2 \wedge \vdash\ c_2\ q\ ]\!]$
$\Longrightarrow t \in pre\ c_2 \wedge \vdash (AnnSeq\ c_2\ c_1)\ q$

By case analysis on $\vdash (AnnSeq\ c_0\ c_1)\ q$ we obtain $\vdash\ c_0\ (pre\ c_1)$ and $\vdash\ c_1\ q$ from the *Seq*2 rule. Unfortunately, we also obtain $\vdash (AnnSeq\ c_0\ c_1)\ q'$ and $q' \subseteq q$ from the rule of consequence. This second subgoal is basically the original subgoal. This circular effect is due to the generality of the consequence rule. This rule is so general that it can always be applied. To avoid this, we prove a more appropriate version of the inductive cases principle which applies the consequence rule at most once:

lemma *ann-hoare-case-analysis*: $\vdash C\ q' \Longrightarrow$
$(\forall r\, f.\ C = \mathit{AnnBasic}\ r\ f \longrightarrow (\exists q.\ r \subseteq \{s.\ f\ s \in q\} \wedge q \subseteq q')) \wedge$
$(\forall c_0\ c_1.\ C = \mathit{AnnSeq}\ c_0\ c_1 \longrightarrow (\exists q.\ q \subseteq q' \wedge \vdash c_0\ \mathit{pre}\ c_1 \wedge \vdash c_1\ q)) \wedge$
$(\forall r\ b\ c_1\ c_2.\ C = \mathit{AnnCond}_1\ r\ b\ c_1\ c_2 \longrightarrow (\exists q.\ q \subseteq q' \wedge$
$r \cap b \subseteq \mathit{pre}\ c_1 \wedge \vdash c_1\ q \wedge r \cap -b \subseteq \mathit{pre}\ c_2 \wedge \vdash c_2\ q)) \wedge$
$(\forall r\ b\ c.\ C = \mathit{AnnCond}_2\ r\ b\ c \longrightarrow$
$(\exists q.\ q \subseteq q' \wedge r \cap b \subseteq \mathit{pre}\ c\ \wedge \vdash c\ q \wedge r \cap -b \subseteq q)) \wedge$
$(\forall r\ i\ b\ c.\ C = \mathit{AnnWhile}\ r\ b\ i\ c \longrightarrow$
$(\exists q.\ q \subseteq q' \wedge r \subseteq i \wedge i \cap b \subseteq \mathit{pre}\ c \wedge \vdash c\ i \wedge i \cap -b \subseteq q)) \wedge$
$(\forall r\ b\ c.\ C = \mathit{AnnAwait}\ r\ b\ c \longrightarrow (\exists q.\ q \subseteq q' \wedge \Vdash\ (r \cap b)\ c\ q))$

Using this theorem instead we obtain only $\vdash c_0\ (\mathit{pre}\ c_1)$ and $\vdash c_1\ q$. By instantiating $\forall q.\ \vdash c_0\ q \longrightarrow t \in \mathit{pre}\ c_2 \wedge \vdash c_2\ q$ with $\mathit{pre}\ c_1$ we prove $t \in \mathit{pre}\ c_2$ and obtain $\vdash c_2\ (\mathit{pre}\ c_1)$. The remaining conclusion, $\vdash (\mathit{AnnSeq}\ c_2\ c_1)\ q$, follows from the rule *Seq*2.

WhileT: After some simplification the corresponding subgoal is:

$[\![\ s \in b;\ s \in r;\ \vdash (\mathit{AnnWhile}\ r\ b\ i\ c)\ q;\ r \subseteq i;\ i \cap b \subseteq \mathit{pre}\ c;\ \vdash c\ i;\ i \cap -\ b \subseteq q\ ]\!]$
$\Longrightarrow\ \vdash (\mathit{AnnSeq}\ c\ (\mathit{AnnWhile}\ i\ b\ i\ c))\ q$

After applying *AnnSeq* backwards we obtain two subgoals. The first one

$[\![\ s \in b;\ s \in r;\ \vdash (\mathit{AnnWhile}\ r\ b\ i\ c)\ q;\ r \subseteq i;\ i \cap b \subseteq \mathit{pre}\ c;\ \vdash c\ i;\ i \cap -\ b \subseteq q\ ]\!]$
$\Longrightarrow\ \vdash c\ \mathit{pre}\ (\mathit{AnnWhile}\ i\ b\ i\ c)$

is proven by simplification because $\mathit{pre}\ (\mathit{AnnWhile}\ i\ b\ i\ c) = i$. Observe that this follows from the definition of the semantics rule *AnnWhileT*. We mentioned in §2.2 that the annotations do not play any role in the definition of the rules of the semantics. However, if we wrote r instead of i for that precondition, this subgoal would not be provable. The second subgoal is solved by applying *AnnWhile* backwards.

Await: This is an axiom of the system so there is no induction hypothesis:

$[\![\ s \in b;\ \mathit{atom\text{-}com}\ c;\ (c,\ s) -P{*}{\rightarrow}\ (\mathit{Parallel}\ [],\ t);\ s \in r;$
$\vdash (\mathit{AnnAwait}\ r\ b\ c)\ q\ ]\!] \Longrightarrow t \in q$

By case analysis on $\vdash (\mathit{AnnAwait}\ r\ b\ c)$ we obtain $\Vdash (r \cap b)\ c\ q$. The system for atomic programs is sound, thus $\models (r \cap b)\ c\ q$. From $s \in r \cap b$ and the definition of validity for atomic programs we prove $t \in q$. $\square$

Finally, we state the soundness theorem for component programs:

theorem *ann-hoare-sound*: $\vdash c\ q \implies \models c\ q$

The proof is immediate using the *Strong-Soundness* theorem.

2.5.3 Soundness of the System for Parallel Programs

The most interesting result of the soundness proof is the soundness of the *Parallel* rule. Like for component programs, we first show the corresponding stronger property called *strong soundness for parallel programs*.

Intuitively, if the proof outline of each component program satisfies the strong soundness property and the actions of the other components do not "interfere", then every component is able to establish the intended postcondition. Then, if all components finish their computations the final state satisfies all postconditions simultaneously.

For example, consider the standard proof outlines (written in a standard syntax) $\{x = 0\}\ x{:=}\ x{+}1\ \{x = 1\}$ and $\{\mathit{True}\}\ x{:=}\ 0\ \{x = 0\}$. They are obviously correct, but not interference free. For instance, the postcondition $\{x = 0\}$ is not preserved under the execution of $x{:=}\ x{+}1$.

However, if we weaken the postconditions and consider the annotations $\{x = 0\}\ x{:=}\ x{+}1\ \{x = 0 \lor x = 1\}$ and $\{\mathit{True}\}\ x{:=}\ 0\ \{x = 0 \lor x = 1\}$ we obtain both, correct and interference free proof outlines.

Finding annotations for each component that are strong enough to satisfy its specification, and yet weak enough to remain invariant under the execution of all atomic actions of other components is often a difficult task that requires perseverance.

The strong soundness theorem for parallel programs states that whenever flow of control reaches a point annotated by an assertion, this assertion is true. The difference is that in a parallel program the control resides simultaneously at several points. Thus, we have to prove that the assertions attached to those points are simultaneously true.

Strong Soundness Theorem for Parallel Programs

Let *Ts* be a list of pairs formed by (optional) component programs and their postcondition. Suppose that each component $Ts!i$ such that $Ts!i =$

(*Some c*, *q*) for some annotated command *c* and some postcondition *q*, has a derivation in the system *ann-hoare*, i.e. $\vdash c\ q$, and *interfree Ts* also holds.

Assume also that (*Parallel Ts*, *s*) $-P{*}\!\rightarrow$ (*Parallel Rs*, *t*) for some list of component programs *Rs* and some states *s*, *t* such that *s* satisfies the precondition of all component programs *Ts*!*i*. Then, all component programs *Rs*!*j* of *Parallel Rs* satisfy that

- if *com* (*Rs*!*j*) = *Some c* for a command *c*, then $t \in$ *pre c*,
- if *com* (*Rs*!*j*) = *None*, then $t \in$ *post* (*Rs*!*j*).

In particular if *com* (*Rs*!*j*) = *None* for all *j*, we have that $t \in$ *post* (*Rs*!*j*) for all *j* such that $j <$ *length Rs*.

The formal lemma as formulated in Isabelle is:

lemma *Parallel-Strong-Soundness*:
$\llbracket$ (*Parallel Ts*, *s*) $-P{*}\!\rightarrow$ (*Parallel Rs*, *t*); *interfree Ts*; $j <$ *length Rs*;
$\forall i <$ *length Ts*. $\exists c\ q$. *Ts*!*i* = (*Some c*, *q*) $\wedge$ $s \in$ *pre c* $\wedge \vdash c\ q$ $\rrbracket$
$\Longrightarrow$ *if com* (*Rs*!*j*) = *None then* $t \in$ *post* (*Ts*!*j*) *else* $t \in$ *pre* (*the* (*com* (*Rs*!*j*)))

Proof. Like in the case of component programs, the theorem in the above form is too weak. The conclusion must establish two more properties of the reached configuration, namely, that the program fragment *the* (*com* (*Rs*!*j*)) has a derivation in the system and that the list of component programs after the transition still satisfies the interference freedom property, i.e. *interfree Rs*. In other words, we have to prove that derivability of a component's proof outline and the interference freedom of a list of proof outlines are preserved throughout the computation:

lemma *Parallel-Strong-Soundness-aux*:
$\llbracket$ (*Ts'*, *s*) $-P{*}\!\rightarrow$ (*Rs'*, *t*); *Ts'* = (*Parallel Ts*);
$\forall i <$ *length Ts*. $\exists c\ q$. *Ts*!*i* = (*Some c*, *q*) $\wedge$ $s \in$ *pre c* $\wedge \vdash c\ q$; *interfree Ts* $\rrbracket$
$\Longrightarrow \forall$ *Rs*. *Rs'* = (*Parallel Rs*) $\longrightarrow$
($\forall j <$ *length Rs*. (*if com* (*Rs*!*j*) = *None then* $t \in$ *post* (*Ts*!*j*)
else $t \in$ *pre* (*the* (*com* (*Rs*!*j*))) $\wedge \vdash$ *the* (*com* (*Rs*!*j*)) *post* (*Ts*!*j*))) $\wedge$
interfree Rs

The proof is by induction on the length of the computation. If the length is 0, the proof is trivial since $Ts' = Rs'$ and $s = t$. If the length is > 0, then for some list of component programs *Ss* and some state *b*:
(*Parallel Ts*,*s*) $-P{*}\!\rightarrow$ (*Parallel Ss*,*b*) $-P1\!\rightarrow$ (*Parallel* (*Ss*[*i*:=(*co*, *q*)]),*t*)

where the last step is performed by the ith component of *Ss* through transition $(Some\ c,\ b)\ -1\rightarrow\ (co,\ t)$ and $Rs' = Parallel\ (Ss[i := (co,\ q)])$.

The conclusion of the lemma is a conjunction of two clauses. The second one, namely *interfree* $(Ss[i := (co,\ q)])$ is proven with the following lemma:

lemma *interfree-lemma*:
$[\![\ (Some\ c,\ s)\ -1\rightarrow\ (co,\ t);\ interfree\ Ts;\ i < length\ Ts;\ \ Ts!i = (Some\ c,\ q)\ ]\!]$
$\Longrightarrow interfree\ (Ts[i{:=}\ (co,\ q)])$

The proof of this lemma follows from two symmetric properties of the predicate *interfree-aux*, both proven by rule induction on the *ann-transition* relation:

lemma *interfree-aux*1:
$[\![\ (co,\ s)\ -1\rightarrow\ (co',\ t);\ interfree\text{-}aux\ (co_1,\ q,\ co)\ ]\!] \Longrightarrow interfree\text{-}aux\ (co_1,\ q,\ co')$
lemma *interfree-aux*2:
$[\![\ (co,\ s)\ -1\rightarrow\ (co',\ t);\ interfree\text{-}aux\ (co,\ q,\ co_1)\ ]\!] \Longrightarrow interfree\text{-}aux\ (co',\ q,\ co_1)$

For the other clause of the conclusion, two cases arise: $i = j$ or $i \neq j$. The first one means that the transition occurred in the same component that we were observing, i.e. component j. The proof amounts to checking strong soundness of a proof outline which is exactly the *Strong-Soundness* theorem.

The case $i \neq j$ means that the last transition $(Some\ c,\ b)\ -1\rightarrow\ (co,\ t)$ was performed by a component i which is not the one we were observing, i.e. not the fixed j. We must prove that component j fulfills the conclusion of the theorem just the same.

The proof proceeds by rule induction on the last *ann-transition* relation. The corresponding "appropriate" auxiliary lemma is:

lemma *Parallel-Strong-Soundness-aux-aux*:
$[\![\ (Some\ c,\ b)\ -1\rightarrow\ (co,\ t);\ i < length\ Ts;\ com\ (Ts!i) = Some\ c;$
$\quad \forall i < length\ Ts.\ if\ com\ (Ts!i) = None\ then\ b \in post\ (Ts!i)$
$\quad\quad else\ b \in pre\ (the\ (com\ (Ts!i))) \wedge \vdash the\ (com\ (Ts!i))\ post\ (Ts!i);$
$\quad interfree\ Ts;\ j < length\ Ts;\ i \neq j\ ]\!]$
$\Longrightarrow if\ com\ (Ts!j) = None\ then\ t \in post\ (Ts!j)$
$\quad else\ t \in pre\ (the\ (com\ (Ts!j))) \wedge \vdash the\ (com\ (Ts!j))\ post\ (Ts!j)$

If the last step in the computation consists of the evaluation of a Boolean expression, then $b = t$. The proof follows by instantiating the universal quantification in the premise with j.

Otherwise, the last step consists of the execution of a basic action, an atomic region or some transition in a sequential composition of commands. We discuss two of them; the remaining two cases are analogous:

Basic: Suppose the command of the ith component is *AnnBasic r f*, then the last transition is $(\mathit{Some}\ (\mathit{AnnBasic}\ r\ f),\ b)\ {-1\rightarrow}\ (\mathit{None},\ f\ b)$. By assumption $b \in r$. Then,

- If $\mathit{com}\ (\mathit{Ts}!j) = \mathit{None}$, then by assumption $b \in \mathit{post}\ (\mathit{Ts}!j)$. By the interference freedom hypothesis we have $\models (\mathit{post}\ (\mathit{Ts}!j) \cap r)\ \mathit{Basic}\ f\ \mathit{post}\ (\mathit{Ts}!j)$. From the definition of validity and $b \in \mathit{post}\ (\mathit{Ts}!j) \cap r$, we conclude that $f\ b \in \mathit{post}\ (\mathit{Ts}!j)$.

- If $\mathit{com}\ (\mathit{Ts}!j) = \mathit{Some}\ y$ for some command y, we obtain from the assumptions that $b \in \mathit{pre}\ y$. By the interference freedom of *Ts* we have:

  $\forall p \in \mathit{assertions}\ y.\ \models (p \cap r)\ \mathit{Basic}\ f\ p.$

  By structural induction on c we prove the lemma: $\mathit{pre}\ c \in \mathit{assertions}\ c$. Hence, we can instantiate the previous assumption with *pre y* obtaining $\models (\mathit{pre}\ y \cap r)\ \mathit{Basic}\ f\ (\mathit{pre}\ y)$. Finally, from the definition of validity $f\ b \in \mathit{pre}\ y$.

Seq2: Suppose now the command of the ith component is $\mathit{AnnSeq}\ c_0\ c_1$, and the last transition is

$$(\mathit{Some}\ (\mathit{AnnSeq}\ c_0\ c_1),\ b)\ {-1\rightarrow}\ (\mathit{Some}\ (\mathit{AnnSeq}\ c_2\ c_1),\ t)$$

Then, from the *ann-transition* rule *Seq*2 we know that

$$(\mathit{Some}\ c_0,\ b)\ {-1\rightarrow}\ (\mathit{Some}\ c_2,\ t)$$

We instantiate the universally quantified variable *Ts* in the induction hypothesis with $\mathit{Ts}[i := (\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]$. Thereby, we obtain the information we need about t but referring to the above instantiation. Since all components of *Ts* other than the component i remain unchanged, the conclusion of the induction hypothesis is exactly the conclusion of the subgoal. Thus, it remains to be shown that the premises required to validate the conclusion of the induction hypothesis are indeed fulfilled by the instantiation:

lemma *Parallel-Strong-Soundness-Seq*:
$[\![\ \forall i < \mathit{length}\ \mathit{Ts}.\ \mathit{if}\ \mathit{com}\ (\mathit{Ts}!i) = \mathit{None}\ \mathit{then}\ b \in \mathit{post}\ (\mathit{Ts}!i)$
$\quad \mathit{else}\ b \in \mathit{pre}\ (\mathit{the}\ (\mathit{com}\ (\mathit{Ts}!i))) \wedge \vdash \mathit{the}\ (\mathit{com}\ (\mathit{Ts}!i))\ \mathit{post}\ (\mathit{Ts}!i);$

$$
\begin{array}{l}
\quad \mathit{com}\ (\mathit{Ts}!i) = \mathit{Some}\ (\mathit{AnnSeq}\ c_0\ c_1);\ i < \mathit{length}\ \mathit{Ts};\ \mathit{interfree}\ \mathit{Ts}\ \rrbracket \Longrightarrow \\
\ (\forall\, \mathit{ia} < \mathit{length}\ \mathit{Ts}.\ (\mathit{if}\ \mathit{com}\ (\mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]!\mathit{ia}) = \mathit{None} \\
\quad \mathit{then}\ b \in \mathit{post}\ (\mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]!\mathit{ia}) \\
\quad \mathit{else}\ b \in \mathit{pre}\ (\mathit{the}\ (\mathit{com}\ (\mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]!\mathit{ia}))) \\
\wedge \vdash \mathit{the}\ (\mathit{com}\ (\mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]!\mathit{ia}))\ \mathit{post}\ (\mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]!\mathit{ia}))) \\
\wedge\ \mathit{interfree}\ (\mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)])
\end{array}
$$

The proof is fairly straightforward. The only modification concerns component i, i.e. we substitute $(\mathit{Some}\ \mathit{AnnSeq}\ c_0\ c_1,\ q)$ for $(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)$. The postcondition remains the same, and so does the precondition since pre $(\mathit{AnnSeq}\ c_0\ c_1) = \mathit{pre}\ c_0$.

To show $\vdash\ c_0\ \mathit{pre}\ c_1$, we instantiate the assumption for component i obtaining $\vdash (\mathit{AnnSeq}\ c_0\ c_1)\ q$. By the rule for sequential composition $\vdash\ c_0$ $\mathit{pre}\ c_1$ also holds.

At last showing $\mathit{interfree}\ \mathit{Ts} \Longrightarrow \mathit{interfree}\ \mathit{Ts}[i{:=}(\mathit{Some}\ c_0,\ \mathit{pre}\ c_1)]$ is straightforward. This concludes the proof of the *Parallel-Strong-Soundness* theorem. □

The final result is the soundness of the full system of rules for parallel programs:

theorem *oghoare-sound*: $\Vdash p\ c\ q \Longrightarrow \models p\ c\ q$

Soundness of the rule for parallel composition follows directly from the *Parallel-Strong-Soundness* theorem. The proofs of soundness for the remaining inference rules have already been proven in the soundness proof for the system of atomic programs.

Our soundness proof is new with respect to those found in the literature. By including preconditions in the program's syntax we achieve a simpler and more intuitive formulation. The textbook we follow as a model for our formalization, namely [Apt and Olderog, 1991], defines the program syntax devoid of any annotation and attaches the preconditions separately. In order to refer to the precondition reached by the execution of a program S, they define a recursive function *at* such that, given a program S and a subprogram T, *at (T, S)* returns the remainder of S that is to be executed when the control is at subprogram T. Using this function they are able to refer to the precondition of the remaining subprogram. Unfortunately, this function is not well-defined: a program S might contain several identical subprograms T, so that the behavior of *at* is unclear. To resolve these ambiguities they informally propose to attach labels to each basic statement in the program. In contrast, the function *pre* of our formalization is trivially well-defined and devoid of such difficulties.

2.6 Generation of Verification Conditions

Due to the presence of the test for interference freedom, the proof method of Owicki and Gries is not *compositional*, i.e. it does not allow a derivation of a correctness specification of a parallel program from the specifications of its components *without* reference to their internal structure. This causes this method to be very costly in practice. For example, in the case of two component programs of length l_1 and l_2, proving interference freedom requires proving $l_1 \times l_2$ additional correctness formulas. Most of them are trivially satisfied because they check an assignment or atomic region a against an assertion which is disjoint from the variables changed in a. By automating this tedious work the user can be sure that all cases are considered.

Fortunately, Hoare-like methods possess the necessary structure to be automated. The proof rules of the system are syntax directed and can be used to generate the necessary verification conditions by using the rules backwards. This process has been encapsulated in an Isabelle tactic. The generated verification conditions are statements of the logic of assertions devoid of any mention of the programming language. The correctness of the program specification depends upon the validity of these conditions, which can be checked using standard Isabelle proof strategies.

As far as the user is concerned, only the name of the defined tactic is relevant. We call the tactic *oghoare*. It is simply applied to a goal stating that some parallel program's specification (with full proof outlines for the component programs) is derivable in the system with the same name (*oghoare*). As a result a subgoal for each verification condition is generated. A detailed explanation of the design of the tactics can be found in appendix A.

2.7 Concrete Syntax

In the previous sections we used an abstract representation for the syntax of the programming language. In particular, the type of the state was left completely indeterminate. This is convenient for meta-theoretical reasoning about the language, however, in order to apply the method for verification, we need to write real programs. In this section we describe the particular formalization of the state and introduce concrete syntax for commands and assertions that allow us to write programs in a familiar way.

2.7.1 Formalization of the State

The state of a program at some point during execution is usually defined as the tuple of values of program variables (or as a mapping that returns the values of the program variables) at each point during execution. In any case, imperative programs manipulate the state by referencing and assigning program variables. Thus, we need a representation for the state that allows these two operations.

Finding an adequate model for the representation of state spaces has been a tricky issue in the story of formal tools for verification of programs. We briefly describe two of the solutions that have been previously implemented in HOL, and the approach used in this thesis.

State as Tuple

The first one is a rather simple approach proposed by [von Wright *et al.*, 1993]. The state is represented as the tuple of the variables appearing in a particular program and implicitly abstract expressions involving variables over this tuple[2]. For example, suppose we have the annotated program (written in a familiar syntax):

$$\textsf{\textbf{vars}}\ x\ y.\ \{\!|\,\mathit{True}\,|\!\}\ x{:=}0;\ \{\!|\,x = 0\,|\!\}\ y{:=}1$$

The state is then represented by the tuple $(x,\ y)$. The internal representation would be

$$\begin{array}{ll} \mathit{AnnSeq} & (\mathit{AnnBasic}\ \{(x,\ y).\ \mathit{True}\}\ (\lambda(x,\ y).\ (0,\ y))) \\ & (\mathit{AnnBasic}\ \{(x,\ y).\ x = 0\}\ (\lambda(x,\ y).\ (x,\ 1))) \end{array}$$

The explicit declaration of variables **vars** $x\ y$ is important for translation functions in order to distinguish program variables depending on the state from constants of the underlying logic.

With this approach variables can have any name and any type. Moreover, operations, syntax, etc. on their values can be directly inherited from Isabelle/HOL's theories and used in programs.

[2]This was the model that we adopted originally. A previous version of the examples verified with the Owicki-Gries method were carried out using this approach.

The main disadvantage is that variable names are bound, thus they have no first-class existence. In HOL there is no difference between $\lambda(x,\ y).\ x + y$ and $\lambda(s,\ t).\ s + t$. This complicates the translation functions which have to "remember" the original names as given by the explicit declaration to avoid renaming of variables throughout the transformations. Besides, abstraction over tuples is not primitive in HOL. It is achieved by suitable combinations of ordinary abstraction and an uncurrying function of type $(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha \times \beta \Rightarrow \gamma$. A one-to-one translation between input and output syntax is sometimes impossible. For example, if the state of a program is $(x,\ y)$ and the input program contains the dummy assignment $x := x$ the translation into internal syntax will return $\lambda(x,\ y).\ (x,\ y)$. The function that translates from internal into external syntax cannot distinguish whether the original input was $x := x$, or $y := y$.

Another disadvantage is the poor modularity: program fragments can be defined separately, but they can only be put together if they depend on the exact same state tuple.

State as Function

An alternative, originally used in [Gordon, 1989], consists of defining the state as a partial function from variable names to values: *name* $\Rightarrow$ *value option*. This approach gives variables a first-class existence, however, types of variables have to be modeled explicitly. The simplest model would be to require all variables in a program to have the same type, this was the first formalization of program variables in Isabelle/HOL [Galm, 1995]. While this model is maybe enough for simple programs with variables ranging over numbers or booleans it is already insufficient when composed variables like arrays are required. A first way out of this situation is to use an enumerated type containing all the required types. However, at least theoretically, imperative programs can use an unlimited range of types (e.g. arrays of arrays of arrays ...). A better choice is to use a properly *recursive* type. This is the approach used in [Harrison, 1998]. It can be achieved by disjoint sum types or by inductive datatypes as follows:

types α *array* = *nat* $\times$ α *list*
datatype *value* =
  Bool bool
 | *Nat nat*
 | *Array value array*
 | *Pointer value*

Following [Harrison, 1998], arrays are represented as a pair consisting of a starting index and a list of elements. With this model it is possible to have an unlimited range of types built from a fixed set of constructors. The main problem is the need to cope with type structure within the logic. This basically means that type-correctness of programs has to be proved every time. This may be feasible for meta-theoretical studies of a programming language, but quite cumbersome in concrete verification tasks.

State as Record

Finally, a model that has all the advantages of the previous two models is based on a formalization of the state as an Isabelle/HOL record type [Naraschewski and Wenzel, 1998]. This type automatically supplies selecting and updating functions for each field. This model was first used by Markus Wenzel in his version of the Hoare logic for sequential programs in Isabelle/Isar [Wenzel, 2001b]. Program variables have a first-class existence and can range over any type. Operations on their value domains can be inherited directly from Isabelle/HOL theories.

Program variables must be previously declared as an Isabelle/HOL record type. Each variable is represented by a record field whose type is the value domain of the variable. For example, consider a program with a single variable x ranging over the natural numbers. Before writing the program, we declare the following record:

$$\textbf{record } \mathit{state} = x :: \mathit{nat}$$

Automatically we obtain a selector function: $x :: \mathit{state} \Rightarrow \mathit{nat}$, and an update function: $x\text{-}\mathit{update} :: \mathit{nat} \Rightarrow \mathit{state} \Rightarrow \mathit{state}$ such that the standard properties of record fields hold. This is optimal for our purposes: the selector function x is used to reference the value of x at a certain state, and the update function is used to model assignments to the variable.

As we shall see in the examples throughout this thesis, concrete syntax can be defined in a very elegant way. The basic idea is based on the *quote/antiquote* technique. A *quotation* is an expression which is implicitly abstracted, in our case over the state space. An *antiquotation* is a marked expression (for example by the antiquote symbol '´') within a quotation that refers to the implicit argument, in our case to the state. An antiquotation would select (or even update) components from the state.

The syntax for quoted expressions and antiquoted expressions inside a quotation is:

syntax

$\textit{-quote} \quad :: \beta \Rightarrow (\alpha \Rightarrow \beta) \qquad (\ll\text{-}\gg)$

$\textit{-antiquote} \quad :: (\alpha \Rightarrow \beta) \Rightarrow \beta \qquad (\acute{}\text{-})$

A quotation $\ll b \gg$ where b has type β represents a function $\lambda s.\ b$ with type $\alpha \Rightarrow \beta$. Antiquoted expressions appear within a quotation and are preceded by the symbol $\acute{}$. For example, assume f is a function with type $\alpha \Rightarrow \beta$, then $\ll \acute{}f \gg$ is a quotation, i.e. an abstraction over some bound variable s where the function f is antiquoted, i.e. applied to the implicit argument s. Thus, the internal expression is $\lambda s.\ f\ s$.

For the case where a variable has been declared in a record, for example x above, then x is a selector function. If we write $\ll \acute{}x = 0 \gg$ then the quoted expression $\acute{}x = 0$ is delimited by an abstraction $(\lambda s.\ \acute{}x = 0)$. The expression x appears antiquoted so that x is translated as a function that refers to the implicit argument: $(x\ s)$. As a result we obtain the internal expression $\lambda s.\ x\ s = 0$. Let us see now how assignments to variables are modeled with these techniques.

A basic-command represents any state transformation. However, programming languages usually use single assignments of the form $x{:=}e$ where x is a variable and e a expression of the proper type. First, we define external syntax for both annotated and non-annotated basic-commands:

syntax

$\textit{-Assign} :: \textit{idt} \Rightarrow \beta \Rightarrow \alpha\ \textit{com} \qquad (\acute{}\text{-} := \text{-})$

$\textit{-AnnAssign} :: \alpha\ \textit{assn} \Rightarrow \textit{idt} \Rightarrow \beta \Rightarrow \alpha\ \textit{com} \qquad (\text{-}\ \acute{}\text{-} := \text{-})$

On the left side of the assignment we write simply an identifier which stands for the variable name. The variable appears "artificially" antiquoted in order to keep a uniform notation for variables inside the program. On the right side we write the assigned expression. Variables appearing within this expression must be antiquoted. The internal syntax uses the function *-update-name* on syntax trees which, given an argument x, returns *x-update*.

translations

$\acute{}x := a \rightharpoonup \textit{Basic} \ll \acute{}(\textit{-update-name}\ x\ a) \gg$

$r\ \acute{}x := a \rightharpoonup \textit{AnnBasic}\ r \ll \acute{}(\textit{-update-name}\ x\ a) \gg$

As a result of this translation from external into internal syntax, if we write in our program for example $\acute{}x{:=}\acute{}x + 1$, then internally Isabelle turns it into $\textit{Basic}\ (\lambda s.\ s(\!|x := x\ s + 1|\!))$, where $s(\!|x := x\ s + 1|\!)$ represents the record s where the field x has been updated to the value $x\ s + 1$. The defined concrete syntax for assignments does not allow for multiple assignments, but they can obviously be expressed in the abstract syntax.

Assertions are enclosed in special brackets to avoid confusion with set notation.

syntax
$\text{-}\mathit{Assert} :: \alpha \Rightarrow \alpha\ \mathit{set} \quad (\{\!|\text{-}|\!\})$

An object enclosed by an assertion, say b, is a boolean expression (possibly containing antiquoted variable names), which is internally quoted, i.e. abstracted over the state. This function is then passed on as the argument of $\mathit{Collect} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set}$. Internally, $\{\!|b|\!\}$ represents the set of states satisfying the predicate b.

translations
$\{\!|b|\!\} \rightharpoonup \mathit{Collect} \ll b \gg$

Further advantages of this model are:

- Antiquotations mark an expression as dependent on the implicit state abstraction. This expression may be "non-atomic", e.g. composition of functions is allowed. This is useful in chapter 4 where abstraction occurs over pairs of states $(s,\ t)$.
- Isabelle/HOL record types may be extended in a linear fashion. For example, if we verify a program that uses a variable x ranging over naturals we declare the record:

  record $\mathit{program}_1 = x :: \mathit{nat}$

  If later we wish to verify a second program with variables x ranging over naturals and b of boolean type, it suffices to declare the extended record:

  record $\mathit{program}_2 = \mathit{program}_1 + b :: \mathit{bool}$

  This is useful for proving derivability of a program by first proving it separately for its subprograms.
- We can also define abbreviations for parts of assertions, parts of programs, etc. as functions over the record type. Such expressions might depend on the values of the program variables in a fixed way. For

example, assume a program with variables x, y and z declared in a record called *vars* such that the expression $\acute{}x + \acute{}y - \acute{}z = \acute{}x - \acute{}y + \acute{}z$ appears in many assertions. We can define a predicate $P :: vars \Rightarrow bool$ over the record type as the quotation $\ll \acute{}x + \acute{}y - \acute{}z = \acute{}x - \acute{}y + \acute{}z \gg$ and then simply write $\acute{}P$ in the assertions.

However, as we shall see in the examples in chapter 3 some expressions over the program variables appear repeatedly but do not always depend on a fixed form of the variables. For example, consider the previous example where the expression $\acute{}x + \acute{}y - \acute{}z = \acute{}x - \acute{}y + \acute{}z$ appears sometimes like that and sometimes as $(\acute{}x + 1) + \acute{}y - \acute{}z = (\acute{}x + 1) - \acute{}y + \acute{}z$. Then, we can define the predicate P as a function over the record-state with a parameter for the value of the variable $\acute{}x$, i.e. the type of P would be $vars \Rightarrow nat \Rightarrow bool$ and its definition $P \equiv \ll \lambda x.\ x + \acute{}y - \acute{}z = x - \acute{}y + \acute{}z \gg$. Then, we can write $\acute{}P\ \acute{}x$ or $\acute{}P\ (\acute{}x + 1)$ depending on the kind of occurrence in the assertions. For large programs with many variables where the predicates in the assertions are frequently repeated, these abbreviations allow us to write clear and short annotations.

With the previous method of representing the state via a tuple of bound variables [von Wright *et al.*, 1993], abbreviations could also be declared as functions over the types of the variables concerned. For example, the predicate P of the previous example would be defined as $P \equiv \lambda(x, y, z).\ x + y - z = x - y + z$. However, every time the predicate is used in an assertion, the arguments have to be written, i.e. $P\ (x, y, z)$ or $P\ (x + 1, y, z)$. When the predicate depends on many variables which appear always in a fixed form, the abbreviations themselves can be unnecessary long.

The many advantages of the representation of program variables used in this thesis will be clearly illustrated in the examples presented here.

2.7.2 Concrete Syntax for Commands and Assertions

In this section we introduce the concrete syntax for commands and assertions in a recipe style. For the reader interested in understanding the examples shown in this thesis and maybe also interested in using the formalization as a verification tool, it suffices to read the rest of this section. The formal specification of the syntax and the corresponding translations are shown in the appendix B.

Variables in the program code appear always marked by an antiquote symbol (´). For example, the variable x is written $´x$ in the program text and inside assertions. In this thesis however, we automatically substitute "antiquoted" variables by the variable name in *sans serif* font (looks nicer on paper). For example, the variable $´x$, is written x in the program text and inside assertions in this thesis.

Assertions are written as boolean expressions (predicates) enclosed between the brackets '$\{\!|$' and '$|\!\}$'. Thus, if we write $\{\!|r|\!\}$ it is internally translated as the set of states satisfying the predicate r. Boolean conditions for if- while- or await-statements are written as normal predicates without any special marking.

Abbreviations for predicates used frequently in the assertions of a program can be given an abbreviation by stating the equality in the premises or defining it previously via **constdefs**. They are defined as abstractions over the state (quoted expressions). Inside assertions they appear antiquoted, i.e. in *sans serif* font.

Table 2.4 shows an overview of the external syntax. Each constructor of the abstract syntax is given a concrete representation. Some commands, separated by a horizontal line in the table, are simply abbreviations for special cases of the commands declared in the abstract syntax. They can be defined by declaring syntax and one-to-one translations (see appendix B). The "new" commands introduced this way are:

Skip is a basic-command whose state-transformation function is the identity.

Atomic regions are *AnnAwait*-statements where the waiting condition is *True*. They appear enclosed in angle brackets $\langle$ and $\rangle$.

Wait -statements are *AnnAwait*-statements where only the waiting condition is important, i.e. the body is *Skip*.

$If-then$ -statements for commands of type α *com* can be defined by a translation with the else-part being *Skip*.

A minor problem appears if we try to define the same syntax for the sequential composition of programs at both layers. Since both have two arguments, Isabelle cannot solve the ambiguity. Thus, we define the syntax at each level slightly different. In the case of annotated programs, sequential composition of c and c' is denoted by $c;;\ c'$, this choice avoids clashes with the predefined ; in Isabelle. For programs of type α *com* the sequential composition of two commands c and c' is denoted by $c,,\ c'$.

For parallel programs there is some concrete syntax of the form

cobegin c_0 $\{\!|q_0|\!\}$ $\|$ $\ldots$ $\|$ c_n $\{\!|q_n|\!\}$ **coend**

for the case where a given number n of component programs are composed in parallel. We also define concrete syntax for program schemas, where the number of components n is a parameter, such as

$A := A[0 := 0] \parallel \ldots \parallel A{:=}\ A[n{-}1 := 0]$

which sets to 0 the components 0 to $n - 1$ in the array A, where arrays are usually modeled as lists. Although the syntax of the programming language does not cater for "...", HOL does. Using the well-known function *map* and the construct $[i..j(]$, which represents the list of natural numbers from i to $j{-}1$, we can express the above schematic program in HOL as follows

$\mathit{Parallel}\ (\mathit{map}\ (\lambda i.\ \{i < \mathit{length}\ A\}\ A := A[i{:=}0]\ \{A!i = 0\})\ [0..n(])$

where the necessary annotations to prove the triple

$\Vdash \{n < \mathit{length}\ A\}\ \mathit{Parallel}\ \ldots\ \{\forall\, i{<}n.\ A!i = 0\}$

have already been inserted.

With the defined concrete syntax for parameterized programs the example above would be written as

cobegin
scheme $[0 \le i < n]$ $\{i < \mathit{length}\ A\}$ $A := A[i{:=}0]$ $\{A!i{=}\ 0\}$
coend

Schematic programs can also appear in parallel with other component programs in the same **cobegin-coend** environment. The index i ranges between the limits indicated in [- $\le i <$-]. Note that i is a bound variable.

In the next section, devoted to the verification of concrete examples, we shall see a sample of all program features presented using the concrete syntax.

2.8 Examples

We have verified all the relevant examples in [Apt and Olderog, 1991]. This section presents solutions to the mutual exclusion, parallel zero search and the producer/consumer problems. Two of the programs for mutual exclusion handle the problem for a non-fixed number of components. In the

Assertion	$\{\!\| r \|\!\}$
Non-annotated commands	
Basic	$\mathsf{x} := e$
Seq	$a_0;, a_1$
Cond	**if** b **then** a_0 **else** a_1 **fi**
While	**while** b **inv** $\{\!\| inv \|\!\}$ **do** a **od**
Skip	**skip**
Cond$_2$	**if** b **then** a_0 **fi**
Annotated commands	
AnnBasic	$\{\!\| r \|\!\}$ $\mathsf{x} := e$
AnnSeq	$c_0;; c_1$
AnnCond$_1$	$\{\!\| r \|\!\}$ **if** b **then** c_1 **else** c_2 **fi**
AnnCond$_2$	$\{\!\| r \|\!\}$ **if** b **then** c **fi**
AnnWhile	$\{\!\| r \|\!\}$ **while** b **inv** $\{\!\| inv \|\!\}$ **do** c **od**
AnnAwait	$\{\!\| r \|\!\}$ **await** b **then** a **end**
AnnSkip	$\{\!\| r \|\!\}$ **skip**
AnnAtom	$\{\!\| r \|\!\}$ $\langle\ a\ \rangle$
AnnWait	$\{\!\| r \|\!\}$ **wait** b **end**
Parallel commands	
Parallel	**cobegin** c_0 $\{\!\| q_0 \|\!\}$ $\|$ $\ldots$ $\|$ c_n $\{\!\| q_n \|\!\}$ **coend**
Schematic	**scheme** $[j \leq i < k]$ c $\{\!\| q \|\!\}$
Convention	
x: program variable	
e: expression of the type of x	
r, b, inv, q, q_i: boolean expressions	
a, a_0, a_1: non-annotated commands	
c, c_0, c_1: annotated commands	
j, k: limits for indexing the component programs	

Table 2.4: Concrete syntax for programs.

next chapter we present a more involved case study of a parameterized program, namely, the verification of a parallel garbage collection algorithm for n mutators. All of them have been verified using the vcg-tactic to generate the verification conditions and standard Isabelle automatic tactics to prove them.

The examples use array variables which are modeled as lists. Access to components of arrays, usually written $A\ [i]$, becomes $A!i$; assignments to components of arrays are written $A := A\ [i{:=}e]$, where $A\ [i{:=}e]$ means that the component at index i in the list A has been replaced by e.

2.8.1 Mutual Exclusion

Mutual exclusion algorithms synchronize n processes, $n \geq 2$, which share a resource. Several properties are expected to be satisfied. The mutual exclusion property guarantees that never more than one process uses the common resource at a time, i.e. only one process may be inside its critical section at each moment. Other properties like deadlock-freedom, defined in §2.1.1, or fairness, which means that all the components get "fair" turns to perform steps, cannot be directly verified in the actual formalization.

Each process S_i in a mutual exclusion algorithm is an infinite loop of the form:

```
$S_i \equiv$ while $True$ do
       $NC_i$;          (non-critical section)
       $ACQ_i$;         (acquire protocol)
       $CS_i$;          (critical section)
       $REL_i$;         (release protocol)
     od
```

We consider a parallel program

$$S \equiv INIT,,\ \textbf{cobegin}\ S_1 \| \ldots \| S_n\ \textbf{coend}$$

where $INIT$ is a loop free program in which the variables used in ACQ_i and REL_i are initialized.

We prove correctness of three solutions to this problem: a *busy wait solution*, i.e. without synchronization, for a two-process algorithm, a second solution using semaphores via synchronization constructs, and the so-called ticket algorithm. The last two examples work for n-processes waiting to access the critical region.

A Busy Wait Solution

A mutual exclusion algorithm where the "adquire protocol" part for each process ACQ_i is of the form T_i; **while** b_i **do skip od** with T_i loop free, is called a *busy wait solution* and the loop **while** b_i **do skip od** is called a *busy wait loop*.

The following such solution to the mutual exclusion problem for two processes is due to [Peterson, 1981]:

```
record Busy-wait-mutex =
 flag₁ :: bool
 flag₂ :: bool
 turn :: nat
 after₁ :: bool
 after₂ :: bool
```

```
lemma Busy-wait-mutex:
 ⊩ {| True |}
 flag₁ := False,, flag₂ := False,,
 cobegin
 {| ¬flag₁ |}
  while True
   inv {| ¬flag₁ |}
  do {| ¬flag₁ |} ⟨flag₁ := True,, after₁ := False⟩;;
     {| flag₁ ∧ ¬after₁ |} ⟨turn := 1,, after₁ := True⟩;;
     {| flag₁ ∧ after₁ ∧ (turn = 1 ∨ turn = 2) |}
      while ¬(flag₂ ⟶ turn = 2)
       inv {| flag₁ ∧ after₁ ∧ (turn = 1 ∨ turn = 2) |}
      do {| flag₁ ∧ after₁ ∧ (turn = 1 ∨ turn = 2) |} skip od;;
     {| flag₁ ∧ after₁ ∧ (flag₂ ∧ after₂ ⟶ turn = 2) |}
      flag₁ := False
  od
 {| False |}
 ‖
 {| ¬flag₂ |}
  while True
   inv {| ¬flag₂ |}
  do {| ¬flag₂ |} ⟨flag₂ := True,,after₂ := False⟩;;
    {| flag₂ ∧ ¬after₂ |} ⟨turn := 2,,after₂ := True⟩;;
    {| flag₂ ∧ after₂ ∧ (turn = 1 ∨ turn = 2) |}
    while $\neg$(flag$_1$ $\longrightarrow$ turn = 1)
      inv $\{\!|$ flag$_2$ $\wedge$ after$_2$ $\wedge$ (turn = 1 $\vee$ turn = 2) $|\!\}$
    do $\{\!|$ flag$_2$ $\wedge$ after$_2$ $\wedge$ (turn = 1 $\vee$ turn = 2) $|\!\}$ skip od;;
   $\{\!|$ flag$_2$ $\wedge$ after$_2$ $\wedge$ (flag$_1$ $\wedge$ after$_1$ $\longrightarrow$ turn = 1) $|\!\}$
    flag$_2$ := $False$
  od
$\{\!|$ $False$ $|\!\}$
coend
$\{\!|$ $False$ $|\!\}$
```

The boolean variable flag_i is set to true when the component S_i intends to enter its critical section. The variable turn is used to manage conflicts, which appear when both components intend to enter their critical sections at the same time. The component which sets the variable turn first is delayed in a busy wait loop. The auxiliary variables after_i indicate whether the assignment turn := i in ACQ_i has been executed.

The critical sections CS_i proceed after the busy wait loop. The preconditions of the critical sections represent the set of states reachable at those points. Observe that program control cannot be ready to enter both critical sections at the same time because no state can satisfy both assertions simultaneously.

The vcg-tactic *oghoare* generates a total of 122 verification conditions, all of them are automatically proven by the Isabelle tactic *auto*, which solves all subgoals simultaneously.

A Solution Using Semaphores

The next solution to the mutual exclusion problem is due to Dijkstra [Dijkstra, 1968]. The algorithm can be generalized to the case where n processes compete to enter their critical sections. It has the simple form:

$$S \equiv \ \mathsf{out} := True,, \ \textbf{cobegin}\ S_1 \| \ldots \| S_n\ \textbf{coend}$$

Written with the formalized syntax for schematic (parameterized) programs (cf. §2.7) the corresponding specification is:

record *Semaphores-parameterized-mutex* =
  out :: *bool*
  who :: *nat*

lemma *Semaphores-parameterized-mutex*: $0 < n \Longrightarrow$

```
$\Vdash$ $\{\!| \ True \ |\!\}$
out := $True$,,
cobegin
  scheme $[0 \le i < n]$
    $\{\!| \ True \ |\!\}$
     while $True$ inv $\{\!| \ True \ |\!\}$
      do $\{\!| \ True \ |\!\}$ await out then out := $False$,, who := $i$ end;;
           $\{\!| \ \neg$out $\wedge$ who $= i \ |\!\}$ out := $True$
      od
   $\{\!| \ False \ |\!\}$
coend
 $\{\!| \ False \ |\!\}$
```

This algorithm uses binary semaphores as a synchronization primitive. A *binary semaphore* is a semaphore that can only take two values. The standard operations on semaphores can be implemented via the synchronization constructor *AnnAwait*. In the program the variable out is a binary semaphore that indicates whether all processes are out of their critical sections. The auxiliary variable who serves to indicate which component, if any, is inside the critical section.

The critical section would be after the assertion $\{\!| \ \neg$ out $\wedge$ who $= i \ |\!\}$. It is easy to see that this precondition cannot hold simultaneously for two different components. Thus, the mutual exclusion property holds.

The vcg-tactic *oghoare* generates 20 verification conditions which are all solved automatically by *auto*.

The Ticket Algorithm

The next example is also a mutual exclusion algorithm for n processes. We prove its correctness based on the proof outline given in [de Roever *et al.*, 2000].

Predicates that are often used in the assertions of a program can be given an abbreviated name by stating the equality in the premises. For reasons concerning the translations between internal and external syntax, the abbreviated expressions appear enclosed between '$\ll$' and '$\gg$'. Then the given name (*Inv* in the next example) appears in *sans serif* font in the assertions, i.e. Inv. These abbreviations are not to be confused with program variables, the reason why they both have the same representation in the program, namely the same font, lies in the fact that both, variables and abbreviations, depend upon the program state.

```
record Ticket-mutex =
 num :: nat
 nextv :: nat
 turn :: nat list
 ind :: nat

lemma Ticket-mutex:
 ⟦ 0 < n; Invariant= « n = length turn ∧ 0 < nextv ∧
  (∀ k < n. ∀ l < n. k ≠ l ⟶ turn!k < num ∧ (turn!k = 0 ∨ turn!k ≠ turn!l)) » ⟧
⟹
⊩ ⦃ n = length turn ⦄
 ind := 0,,
 while ind < n
   inv ⦃ n = length turn ∧ (∀ i < ind. turn!i = 0) ⦄
 do turn := turn [ind := 0],, ind := ind + 1 od,,
 num := 1,, nextv := 1 ,,
 cobegin
   scheme [0 ≤ i < n]
    ⦃ Invariant ⦄
    while True inv ⦃ Invariant ⦄
    do ⦃ Invariant ⦄ ⟨turn := turn [i := num],, num := num + 1⟩;;
        ⦃ Invariant ⦄ wait turn!i = nextv end;;
        ⦃ Invariant ∧ turn!i = nextv ⦄ nextv := nextv + 1
    od
   ⦃ False ⦄
 coend
 ⦃ False ⦄
```

The critical section would be entered before the assignment to nextv, i.e. at the moment of entering the assertion ⦃ Invariant ∧ turn!i=nextv ⦄ holds. The mutual exclusion property is guaranteed because the conjunction of two or more assertions with different values for i before entering the critical section implies *false*. These assertions represent the possible states at that point, consequently, the set of states from which more than one component could enter the critical section is empty.

The application of the vcg-tactic returns 35 subgoals. Simplification tactics leave 11 verification conditions unsolved. Their proofs only need to be further directed by several case distinctions as hinted in the pencil and paper proof of [de Roever *et al.*, 2000].

2.8.2 Parallel Zero Search

The next example is a program that finds a zero of a function f from naturals to naturals, searching in parallel for zeroes that are bigger or smaller than a certain natural a.

```
record Zero-search =
  turn :: nat
  found :: bool
  x :: nat
  y :: nat
```

```
lemma Zero-search:
 $[\![ I_1 = \ll a \leq x \wedge (found \longrightarrow (a < x \wedge f(x) = 0) \vee (y \leq a \wedge f(y) = 0))$
          $\wedge (\neg found \wedge a < x \longrightarrow f(x) \neq 0) \gg ;$
   $I_2 = \ll y \leq a + 1 \wedge (found \longrightarrow (a < x \wedge f(x) = 0) \vee (y \leq a \wedge f(y) = 0))$
          $\wedge (\neg found \wedge y \leq a \longrightarrow f(y) \neq 0) \gg ]\!] \Longrightarrow$
$\Vdash \{\!| \exists u.\ f(u) = 0 |\!\}$
turn := 1,, found := False,,
x := a,, y := a + 1 ,,
cobegin $\{\!| I_1 |\!\}$
  while ¬found
    inv $\{\!| I_1 |\!\}$
  do $\{\!| a \leq x \wedge (found \longrightarrow y \leq a \wedge f(y) = 0) \wedge (a < x \longrightarrow f(x) \neq 0) |\!\}$
     wait turn = 1 end;;
     $\{\!| a \leq x \wedge (found \longrightarrow y \leq a \wedge f(y) = 0) \wedge (a < x \longrightarrow f(x) \neq 0) |\!\}$
     turn := 2;;
     $\{\!| a \leq x \wedge (found \longrightarrow y \leq a \wedge f(y) = 0) \wedge (a < x \longrightarrow f(x) \neq 0) |\!\}$
     ⟨x := x + 1,, if f(x) = 0 then found := True else skip fi⟩
  od;;
  $\{\!| I_1 \wedge found |\!\}$
 turn := 2
  $\{\!| I_1 \wedge found |\!\}$
‖
  $\{\!| I_2 |\!\}$
  while ¬found
    inv $\{\!| I_2 |\!\}$
  do $\{\!| y \leq a + 1 \wedge (found \longrightarrow a < x \wedge f(x) = 0) \wedge (y \leq a \longrightarrow f(y) \neq 0) |\!\}$
    wait turn=2 end;;
    $\{\!| y \leq a + 1 \wedge (found \longrightarrow a < x \wedge f(x) = 0) \wedge (y \leq a \longrightarrow f(y) \neq 0) |\!\}$
    turn := 1;;
    $\{\!|$ y $\leq a + 1 \wedge$ (found $\longrightarrow a <$ x $\wedge f($x$) = 0) \wedge ($y $\leq a \longrightarrow f($y$) \neq 0)$ $|\!\}$
    $\langle$y := y $- 1$,, **if** $f($y$) = 0$ **then** found := *True* **else skip fi**$\rangle$
  **od**;;
  $\{\!|$ I$_2$ $\wedge$ found $|\!\}$
  turn := 1
  $\{\!|$ I$_2$ $\wedge$ found $|\!\}$
**coend**
$\{\!|$ $f($x$) = 0 \vee f($y$) = 0$ $|\!\}$
```

The tactic generates 98 verification conditions. After applying the general automatic tactic *auto*, 40 subgoals remain[3]. They are all proven with the tactic *arith* that automatically solves basic arithmetical problems.

We verify a second simpler solution to this problem without using synchronization:

```
**lemma** *Zero-Search*$_2$:
$[\![$ $I_1 = \ll a \leq$ x $\wedge$ (found $\longrightarrow (a <$ x $\wedge f($x$) = 0) \vee ($y $\leq a \wedge f($y$) = 0))$
          $\wedge$ ($\neg$found $\wedge a <$ x $\longrightarrow f($x$) \neq 0) \gg$;
  $I_2 = \ll$ y $\leq a + 1 \wedge$ (found $\longrightarrow (a <$ x $\wedge f($x$) = 0) \vee ($y $\leq a \wedge f($y$) = 0))$
          $\wedge$ ($\neg$found $\wedge$ y $\leq a \longrightarrow f($y$) \neq 0) \gg$ $]\!] \Longrightarrow$
$\Vdash$ $\{\!|$ $\exists u.\ f(u) = 0$ $|\!\}$
found := *False*,,
x := $a$,, y := $a + 1$,,
**cobegin** $\{\!|$ I$_1$ $|\!\}$
 **while** $\neg$found
   **inv** $\{\!|$ I$_1$ $|\!\}$
 **do** $\{\!|$ $a \leq$ x $\wedge$ (found $\longrightarrow$ y $\leq a \wedge f($y$) = 0) \wedge (a <$ x $\longrightarrow f($x$) \neq 0)$ $|\!\}$
    $\langle$x := x $+ 1$,, **if** $f($x$) = 0$ **then** found := *True* **else skip fi**$\rangle$
 **od**
 $\{\!|$ I$_1$ $\wedge$ found $|\!\}$
$\|$
 $\{\!|$ I$_2$ $|\!\}$
 **while** $\neg$found
   **inv** $\{\!|$ I$_2$ $|\!\}$
 **do** $\{\!|$ y $\leq a + 1 \wedge$ (found $\longrightarrow a <$ x $\wedge f($x$) = 0) \wedge ($y $\leq a \longrightarrow f($y$) \neq 0)$ $|\!\}$
    $\langle$y := y $- 1$,, **if** $f($y$) = 0$ **then** found := *True* **else skip fi**$\rangle$
  **od**
```

[3]The tactic *auto* simplifies all subgoals simultaneously and might generate several simpler subgoals out of one.

$\{\!|\ \mathsf{I}_2 \wedge \mathsf{found}\ |\!\}$
coend
$\{\!|\ f(\mathsf{x}) = 0 \vee f(\mathsf{y}) = 0\ |\!\}$

Only 20 verification conditions are generated in this simplified version. The application of *auto* leaves 32 arithmetical subgoals, all solved automatically by the tactic *arith*.

2.8.3 Producer/Consumer

This problem coordinates two processes, producer and consumer, that share a common, bounded buffer. The producer puts information into the buffer, the consumer takes it out. Trouble arises when the producer attempts to put a new item in a full buffer or the consumer tries to remove an item from an empty buffer. Following Owicki-Gries we express the problem as a parallel program with shared variables and await-commands. It copies the elements of an array a into an array variable b. Note that a is not a variable of the program, thus its value cannot be overwritten.

$$\{0 < \mathit{length}\ a \wedge 0 < \mathit{length}\ \mathsf{buffer} \wedge \mathit{length}\ \mathsf{b} = \mathit{length}\ a\}$$
$$\textbf{cobegin}\ \mathit{producer} \parallel \mathit{consumer}\ \textbf{coend}$$
$$\{\forall\, k < \mathit{length}\ a.\ a!k = \mathsf{b}!k\}$$

The precondition imposes that the length of a and b be equal, and a, b and buffer have non-zero length. The full program is shown below:

record *Producer-consumer* =
  ins :: *nat*
  outs :: *nat*
  i :: *nat*
  j :: *nat*
  vx :: *nat*
  vy :: *nat*
  buffer :: *nat list*
  b :: *nat list*

For readability we used some abbreviations that can be defined in the premises of the lemma.

lemma *Producer-consumer*:
  $[\![\ \mathit{INIT} = \ll 0 < \mathit{length}\ a \wedge 0 < \mathit{length}\ \mathsf{buffer} \wedge \mathit{length}\ \mathsf{b} = \mathit{length}\ a \gg;$

```
  $I = \ll (\forall k.\ \mathsf{outs} \le k \wedge k < \mathsf{ins} \longrightarrow a!k = \mathsf{buffer}!(k\ mod\ (length\ \mathsf{buffer})))$
       $\wedge\ \mathsf{outs} \le \mathsf{ins} \wedge \mathsf{ins} - \mathsf{outs} \le length\ \mathsf{buffer} \gg$ ;
  $I_1 = \ll \mathsf{I} \wedge \mathsf{i} \le length\ a \gg$ ;
  $p_1 = \ll \mathsf{I}_1 \wedge \mathsf{i} = \mathsf{ins} \gg$ ;
  $I_2 = \ll \mathsf{I} \wedge (\forall k < \mathsf{j}.\ a!k = \mathsf{b}!k) \wedge \mathsf{j} \le length\ a \gg$ ;
  $p_2 = \ll \mathsf{I}_2 \wedge \mathsf{j} = \mathsf{outs} \gg \rrbracket \Longrightarrow$
 $\Vdash \{\!| \mathsf{INIT} |\!\}$
 ins := 0,, outs := 0,, i := 0,, j := 0,,
cobegin $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} |\!\}$
 while i < $length\ a$
   inv $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} |\!\}$
 do $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} \wedge \mathsf{i} < length\ a |\!\}$
     vx := $a$!i;;
     $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} \wedge \mathsf{i} < length\ a \wedge \mathsf{vx} = a!\mathsf{i} |\!\}$
     wait ins − outs < $length$ buffer end;;
     $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} \wedge \mathsf{i} < length\ a \wedge \mathsf{vx} = a!\mathsf{i}\ \wedge$
       $\mathsf{ins} - \mathsf{outs} < length\ \mathsf{buffer} |\!\}$
     buffer := buffer [ins $mod$ ($length$ buffer) := vx];;
     $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} \wedge \mathsf{i} < length\ a\ \wedge$
       $a!\mathsf{i} = \mathsf{buffer}!(\mathsf{ins}\ mod\ (length\ \mathsf{buffer})) \wedge \mathsf{ins} - \mathsf{outs} < length\ \mathsf{buffer} |\!\}$
     ins := ins + 1;;
     $\{\!| \mathsf{I}_1 \wedge \mathsf{INIT} \wedge \mathsf{i} + 1 = \mathsf{ins} \wedge \mathsf{i} < length\ a |\!\}$
     i := i + 1
 od
 $\{\!| \mathsf{p}_1 \wedge \mathsf{INIT} \wedge \mathsf{i} = length\ a |\!\}$
∥
 $\{\!| \mathsf{p}_2 \wedge \mathsf{INIT} |\!\}$
 while j < $length\ a$
   inv $\{\!| \mathsf{p}_2 \wedge \mathsf{INIT} |\!\}$
 do $\{\!| \mathsf{p}_2 \wedge \mathsf{j} < length\ a \wedge \mathsf{INIT} |\!\}$
    wait outs < ins end;;
    $\{\!| \mathsf{p}_2 \wedge \mathsf{j} < length\ a \wedge \mathsf{outs} < \mathsf{ins} \wedge \mathsf{INIT} |\!\}$
    vy := buffer!(outs $mod$ ($length$ buffer));;
    $\{\!| \mathsf{p}_2 \wedge \mathsf{j} < length\ a \wedge \mathsf{outs} < \mathsf{ins} \wedge \mathsf{vy} = a!\mathsf{j} \wedge \mathsf{INIT} |\!\}$
    outs := outs + 1;;
    $\{\!| \mathsf{I}_2 \wedge \mathsf{j} + 1 = \mathsf{outs} \wedge \mathsf{j} < length\ a \wedge \mathsf{vy} = a!\mathsf{j} \wedge \mathsf{INIT} |\!\}$
    b := b [j := vy];;
    $\{\!| \mathsf{I}_2 \wedge \mathsf{j} + 1 = \mathsf{outs} \wedge \mathsf{j} < length\ a \wedge a!\mathsf{j} = \mathsf{b}!\mathsf{j} \wedge \mathsf{INIT} |\!\}$
    j := j + 1
 od
```

$\{\!\!\{\ \mathsf{p}_2 \wedge \mathsf{j} = \mathit{length}\ a \wedge \mathsf{INIT}\ \}\!\!\}$
coend
$\{\!\!\{\ \forall k < \mathit{length}\ a.\ a!k = \mathsf{b}!k\ \}\!\!\}$

Both components share the variables ins and outs, which count the values added to the buffer and the values removed from the buffer, respectively. Thus, the buffer contain ins − outs values at each moment. Expressions ins *mod* (*length* buffer) and outs *mod* (*length* buffer) determine the subscript of the buffer element where the next value is to be added or removed.

The verification of this problem involves proving a total of 138 conditions. Half of them are trivially solved since they refer to triples of the form $(A \cap \mathit{pre}\ r)\ r\ A$ where the atomic action r does not change the variables in A. The rest are automatically solved by Isabelle standard simplification tactics.

2.9 Summary

We have presented the first formalization of the Owicki-Gries method in a general purpose theorem prover. This method can be considered a classic and has been studied extensively since its introduction in 1975 (cf. [Dijkstra, 1976, Apt, 1981a, Apt and Olderog, 1991, Best, 1996, de Roever *et al.*, 2000]). Nevertheless, our formalization yields two main new contributions:

1. A simpler and more intuitive soundness proof with no explicit reference to program locations.

2. A generalized proof rule for parallel composition that allows direct verification of parameterized parallel programs in the system.

In addition, we provide the following features that turn the formalization into a tool suitable for real program verification:

1. Familiar concrete syntax for writing programs like they are usually found in the literature, and

2. A tactic that automatically generates all the verification conditions.

So far we have only verified typical examples from the literature. The next chapter presents the verification of two garbage collection algorithms. These examples better illustrate the applicability of the formalization for two reasons: first, they are larger and more involved programs and second, no complete Owicki-Gries proof existed in the literature. We believe that the availability of the tool was decisive in the search for successful proof outlines.

Chapter 3

Case Study: Single and Multi-Mutator Garbage Collection Algorithms

In this chapter we show that the Owicki-Gries method and its mechanization can be successfully applied to larger examples than those considered in §2.8. We study two incremental garbage collection algorithms, the second one parametric in the number of mutators. These algorithms are particularly tricky and very distinguished scientists have published flawed proofs, some of which were first detected by mechanization attempts. An excellent account of these flaws can be found in [Russinoff, 1994].

We first verify Ben-Ari's classic algorithm [Ben-Ari, 1984]. A pencil and paper proof using the Owicki-Gries method plus ad-hoc reasoning was presented in [van de Snepscheut, 1987]. Our proof follows [van de Snepscheut, 1987], but it manages to formulate the extra reasoning within the Owicki-Gries method. Ben-Ari's algorithm has also been mechanically proven using the Boyer-Moore prover [Russinoff, 1994] and PVS [Havelund, 1996], but none of these proofs uses Owicki-Gries. This makes the algorithm an excellent example for comparing Owicki-Gries with other methods, and for comparing Isabelle/HOL with other theorem provers.

In §3.4 we verify a parameterized garbage collector in which an arbitrary number of mutators work in parallel. This implies that the correctness proof must be carried out for an infinite family of algorithms, which introduces an additional difficulty.

The first extension of Ben-Ari's algorithm to several mutators was published in [Pixley, 1988]. We verify an improved version from [Jonker, 1992] which is finer-grained, uses less colors and has less overhead for the mutators. The author of [Jonker, 1992] gives a proof of correctness using an ad-hoc technique based on observing the behavior of appropriate variant functions. In the same paper it is argued that the Owicki-Gries method is not suitable for this problem. To our knowledge this is the first Owicki-Gries proof and the first mechanized proof of this algorithm.

Thanks to Isabelle's facilities in dealing with concrete syntax, the formalization can be done in a very natural way, where the algorithms and lemmas are as readable as in the original papers.

The chapter is structured as follows: the basics of garbage collection algorithms are described in section 3.1. In section 3.2 we formalize a model for computer memory. Section 3.3 presents the proof of Ben-Ari's algorithm in detail. Section 3.4 presents the proof of the parametric algorithm. In both cases a safety property stating that only garbage nodes are collected is verified. In section 3.5 we compare our proofs with other related works and draw conclusions.

3.1 Incremental Garbage Collection

Garbage collection is the automatic reclamation of memory space[1]. User processes, called *mutators*, might produce garbage while performing their computations. The *collector*'s task is to identify this garbage and to recycle it for future use by appending it to the *free list*. *Incremental* (also called on-the-fly) garbage collection systems are those where the garbage collection work is randomly interleaved with the execution of instructions in the running programs. This is important for real-time applications where memory management operations should never halt the executing program for more than a very brief period.

The *memory* is modeled as a finite directed graph with a fixed number of nodes, where each node has a fixed set of outgoing edges. A predetermined subset of nodes, called the *roots*, is always accessible to the running program. A node is called *reachable* or *accessible* if a directed path exists along the edges from at least one root to that node, otherwise, it is called *garbage*. For marking purposes, each node is associated a color, which can be black or white. The memory structure can only be modified by one of the following three operations: redirect an edge from a reachable node towards a reachable node, append a garbage node to the free list, or change the color of a node.

[1]An excellent survey about garbage collection algorithms can be found in [Wilson, 1992].

The mutators abstractly represent the changes that user programs produce on the memory structure. It is assumed that they only work on nodes that are reachable, having the ability to redirect an edge to some new target. To make garbage collection safe, the mutators cooperate with the collector by assuming the overhead of blackening the new target. Thus, a mutator repeatedly redirects some edge R to some reachable node T, and then colors the node T black.

It is customary to describe the collector's task in this way: identify the nodes that are garbage, i.e. no longer reachable, and append them to the free list, so that their space can be reused by the running program. We abstract from the particular implementation of the free list and simply assume that the collector makes garbage nodes accessible again: since the mutator has the ability to redirect arbitrary accessible edges, it may reuse these nodes. In the sequel adding a node to the free list just means making it accessible.

The collector repeatedly executes two phases, traditionally called *marking phase* and *sweep* or *appending phase*.

During the marking phase, the collector traverses the graph, starting by blackening the set of roots, and marks accessible nodes by coloring them black. This process finishes when all reachable nodes are black. During the appending phase the memory is swept, appending all unmarked (garbage) nodes to the free list. The outline of the algorithms is:

- Marking phase:

    1. Color the roots black.
    2. Visit each edge. If the source is black, color the target black.
    3. Count the black nodes.
    4. If not all reachable nodes are black, go to step 2.

- Appending phase:

    5. Visit each node. If it is white, append it to the free list; if it is black, color it white.

The safety property we prove says that *no reachable node is garbage collected*. In other words, if during the appending operation a node is white, then it is garbage. Clearly, this property holds if step 4 is correct. But how do we determine that all reachable nodes are black? In the case of one mutator, Ben Ari's solution is to keep the result of the last count, and compare it with the result of the current count. If they coincide, then all reachable nodes are black. For n mutators, we compare the results of the last $n+1$ counts. So the algorithms for one and several mutators differ only in step 4.

3.2 Formalization of the Memory

The memory is formalized using two lists of fixed size. The first list, represented by the variable M in the algorithms, has an index for each memory node, i.e. nodes are referred to by natural numbers that range from 0 to *length* $\mathsf{M} - 1$; the color of node i can be consulted by accessing the contents of index i, which in Isabelle's list notation is written as $\mathsf{M}!i$. The datatype *node* is the color of a node, which can be black or white.

datatype *node* = *Black* | *White*

The list of nodes M has the type

types *nodes* = *node list*

The second list, which we refer to by E in the algorithms, models the edges, which are numbered by the indices of the list; each edge is a pair of natural numbers corresponding to the source and the target nodes

types
  edge = *nat* × *nat*
  edges = *edge list*

and *Roots* is an arbitrary set of nodes

consts *Roots* :: *nat set*

Figure 3.1 shows an example of a memory where the set of roots is $\{1,\ 2\}$, the list M of nodes is [*White*, *Black*, *White*, *White*, *White*] and the list E of edges is [(0, 0), (3, 4), (1, 2), (2, 3), (4, 2)].

We define some sets and predicates that are frequently used in the annotations. *Blacks* of a list of nodes returns the set of nodes that are *Black*. *BtoW* is true of the edges that point from a *Black* node to a *White* node. Finally, given a list of edges e, *Reach* e is the set of nodes reachable from *Roots*, i.e. the *Roots* themselves and those nodes such that there exists a

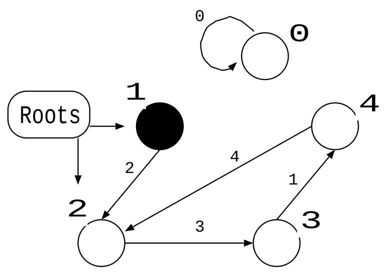

Figure 3.1: An example of the memory.

path along the edges from the node to some root. The formal definitions are:

constdefs

$BtoW :: (edge \times nodes) \Rightarrow bool$
$BtoW \equiv \lambda(e,\, m).\ (m!fst\ e) = Black \wedge (m!snd\ e) \neq Black$

$Blacks :: nodes \Rightarrow nat\ set$
$Blacks\ m \equiv \{i.\ i < length\ m \wedge m!i = Black\}$

$Reach :: edges \Rightarrow nat\ set$
$Reach\ e \equiv \{x.\ x \in Roots \vee$
$(\exists\, path.\ 1 < length\ path \wedge path!(length\ path - 1) \in Roots \wedge x = path!0$
$\wedge (\forall\, i < length\ path - 1.\ (\exists\, j < length\ e.\ e!j = (path!(i + 1),\, path!i))))\}$

The next predicates indicate whether a given set of roots or edges is well-formed:

$Proper\text{-}Roots :: nodes \Rightarrow bool$
$Proper\text{-}Roots\ m \equiv Roots \neq \{\} \wedge Roots \subseteq \{i.\ i < length\ m\}$

$Proper\text{-}Edges :: (nodes \times edges) \Rightarrow bool$
$Proper\text{-}Edges \equiv (\lambda(m,\, e).\ \forall\, i < length\ e.\ fst\ (e!i) < length\ m$
$\wedge\ snd\ (e!i) < length\ m)$

Given a list of nodes, a proper set of roots is a non-empty subset of nodes. Proper edges are those that point from a node to a node, i.e. the first and second components of an edge-pair must be within the range of node indices.

The separate treatment of colors and edges in our data structure is an abstraction that considerably simplifies proofs relating to the changes in the graph. If an edge is redirected, the variable representing the memory, namely M, remains invariant, while coloring does not modify the variable representing the edges E.

Finally, we introduce a last predicate to express that all reachable nodes are black. It is called *Safe* because as we shall see this situation represents a safe state for the memory.

constdefs
  $\mathit{Safe} :: (\mathit{nodes} \times \mathit{edges}) \Rightarrow \mathit{bool}$
  $\mathit{Safe} \equiv \lambda(m,\ e).\ \mathit{Reach}\ e \subseteq \mathit{Blacks}\ m$

3.3 The Single-Mutator Case

We verify van de Snepscheut's version of Ben-Ari's algorithm. We follow the ideas of [van de Snepscheut, 1987], but formulate the proof completely within the Owicki-Gries system.

The program consists of two components, namely, the *Mutator* and the *Collector*. First, we study each component separately and prove that they achieve their intended task whenever they are executed in isolation. Then, we prove that both components can indeed be executed in parallel without interfering with each other.

In order to use the defined concrete syntax for programs (see 2.7), the variables used in the collector and mutator are first declared in a record:

record *gar-coll-state* =
  $M :: \mathit{nodes}$
  $E :: \mathit{edges}$
  $\mathit{bc} :: \mathit{nat\ set}$
  $\mathit{obc} :: \mathit{nat\ set}$
  $\mathit{Ma} :: \mathit{nodes}$
  $\mathit{ind} :: \mathit{nat}$
  $k :: \mathit{nat}$
  $z :: \mathit{bool}$

In the program text and assertions, variables are printed in *sans serif* font. The variable M represents the list of nodes and E the list of edges. The role of the other variables will be explained in the following sections.

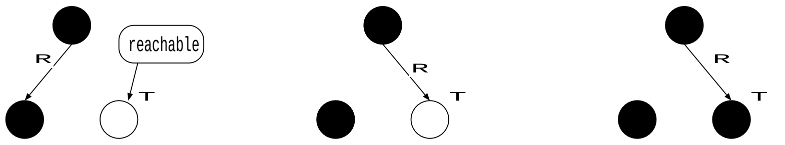

Figure 3.2: The Mutator.

3.3.1 The Mutator

The mutator first redirects an arbitrary edge R from an arbitrary accessible node towards an arbitrary accessible node T. It then colors the new target T black. A graphical description of the actions are shown in figure 3.2. We declare the arbitrarily selected node and edge as constants:

consts
$R :: nat$
$T :: nat$

The following predicate states, given a list of nodes m and a list of edges e, the conditions under which the selected edge R and node T are valid:

constdefs
$\mathit{Mut\text{-}init} :: \mathit{gar\text{-}coll\text{-}state} \Rightarrow \mathit{bool}$
$\mathit{Mut\text{-}init} \equiv \ll T \in \mathit{Reach}\ \mathsf{E} \wedge R < \mathit{length}\ \mathsf{E} \wedge T < \mathit{length}\ \mathsf{M} \gg$

For a more structured proof we have divided the algorithms into modules. A *module* is a piece of code, consisting of one or several instructions. For the mutator we consider two modules, one for each action. An auxiliary variable z is set to false if the mutator has already redirected an edge but has not yet colored the new target. Note that the state is the previously declared record of program variables.

constdefs
$\mathit{Redirect\text{-}Edge} :: \mathit{gar\text{-}coll\text{-}state}\ \mathit{ann\text{-}com}$
$\mathit{Redirect\text{-}Edge} \equiv \{\!|\mathsf{Mut\text{-}init} \wedge \mathsf{z}|\!\}\ \langle \mathsf{E} := \mathsf{E}\ [R := (\mathit{fst}\ (\mathsf{E}!R),\ T)],,\ \mathsf{z} := (\neg \mathsf{z})\rangle$

$\mathit{Color\text{-}Target} :: \mathit{gar\text{-}coll\text{-}state}\ \mathit{ann\text{-}com}$
$\mathit{Color\text{-}Target} \equiv \{\!|\mathsf{Mut\text{-}init} \wedge \neg \mathsf{z}|\!\}\ \langle \mathsf{M} := \mathsf{M}\ [T := \mathit{Black}],,\ \mathsf{z} := (\neg \mathsf{z})\rangle$

$\mathit{Mutator} :: \mathit{gar\text{-}coll\text{-}state}\ \mathit{ann\text{-}com}$
$\mathit{Mutator} \equiv$

```
{|Mut-init ∧ z|}
while True inv {|Mut-init ∧ z|}
do  Redirect-Edge ;; Color-Target od
```

We prove that the mutator's proof outline is correct, by the soundness theorem it suffices to prove that it is derivable in the proof system for component programs. To obtain the full proof outline a postcondition has to be added to the annotated command. Since the program is an infinite loop, no state ever reaches the postcondition.

lemma *Mutator*: $\vdash$ *Mutator* $\{\!|$ *False* $|\!\}$

The verification conditions are generated with the tactic *annhoare*. All are trivially solved except for one which requires the following lemma:

lemma *Graph*1:
$[\![\; t \in Reach\ e;\ r < length\ e \;]\!] \Longrightarrow t \in Reach\ (e\ [r := (fst\ (e!r),\ t)])$

stating that an accessible node cannot be rendered inaccessible by redirecting an edge to it. For the proof it is not necessary to require that the source of the selected edge R be reachable. However, for implementations purposes, it is important that mutators work on nodes that are reachable. Otherwise, the following scenario explained in [Pixley, 1988] could cause problems in the system: Suppose that a certain edge R has been selected by the mutator. If its source becomes unreachable before the redirection is executed, it could be garbage collected. Once it is appended to the free list another process might re-use it resulting in unexpected results when the mutator performs the pending redirection.

3.3.2 The Collector

The collector works in two phases. The first one, called the marking phase first blackens the roots and then executes a loop. The body of the loop consists of first traversing M, coloring all reachable nodes black, and then counting the number of black nodes. The loop terminates if the results of the current count and the previous one coincide. After termination of the loop, the appending phase starts. Here the collector traverses M once more, this time making all white nodes reachable and all black nodes white. Figure 3.3 shows an example of the execution of the collector in isolation.

To structure the proof outline of the collector we define four modules: *Blacken-Roots*, *Propagate-Black*, *Count-Blacks* and *Append*.

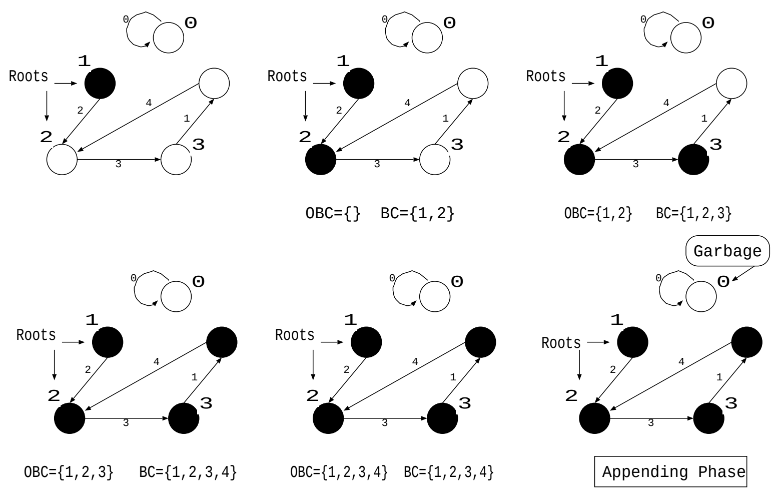

Figure 3.3: The marking phase.

The collector uses, apart form the list of nodes M and the list of edges E, five more variables. bc (black count) and obc (old black count) are used to determine if the set of black nodes has grown during the last *Propagate-Black* phase. Following [van de Snepscheut, 1987], obc is initialized to the empty set, and bc to the set *Roots*[2]. A single auxiliary variable Ma is used for "recording" the value of M after the execution of *Propagate-Black*. This is just an assignment to an auxiliary variable, used exclusively for proof purposes, and therefore not part of the computation. Finally, ind is a counter for loops and k is used to achieve a finer grain of interleaving inside the *Propagate-Black* phase.

consts
Blacken-Roots :: *gar-coll-state ann-com*
Propagate-Black :: *gar-coll-state ann-com*
Count-Blacks :: *gar-coll-state ann-com*
Append :: *gar-coll-state ann-com*

A constant *M-init* is used to give Ma a suitable first value, defined as a list of nodes where only the *Roots* are black.

[2]obc and bc are modeled as sets of black nodes whereas in the original algorithm they represent their cardinalities. We found the set approach more elegant but it simplifies neither the algorithm nor the proofs.

```
consts  M-init :: nodes
constdefs
  Proper-M-init :: nodes ⇒ bool
  Proper-M-init m ≡ Blacks M-init = Roots ∧ length M-init = length m
```

For readability of the assertions we introduce the following abbreviations:

```
constdefs
  Proper :: gar-coll-state ⇒ bool
  Proper ≡ ≪ Proper-Roots M ∧ Proper-Edges (M, E) ∧ Proper-M-init M ≫
```

The proof outline of the collector with modules is:

```
constdefs
  Collector :: gar-coll-state ann-com
  Collector ≡
 {| Proper |}
 while True inv {| Proper |}
 do
    Blacken-Roots;;
   {| Proper ∧ Roots ⊆ Blacks M |} obc := {};;
   {| Proper ∧ Roots ⊆ Blacks M ∧ obc = {} |} bc := Roots;;
   {| Proper ∧ Roots ⊆ Blacks M ∧ obc = {} ∧ bc = Roots |} Ma := M-init;;
   {| Proper ∧ Roots ⊆ Blacks M ∧ obc = {} ∧ bc = Roots ∧ Ma = M-init|}
    while obc ≠ bc
      inv {| Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
             ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ bc ∧ bc ⊆ Blacks M
             ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) |}
    do  {| Proper ∧ Roots ⊆ Blacks M ∧ bc ⊆ Blacks M |}
          obc := bc;;
           Propagate-Black;;
          {| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M
            ∧ bc ⊆ Blacks M ∧ (obc ⊂ Blacks M ∨ Safe (M, E)) |}
          Ma := M;;
          {| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks Ma
            ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M ∧ length Ma = length M
            ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) |}
          bc := {};;
          Count-Blacks
    od;;
    Append
  od
```

Safe (M, E) states that all reachable nodes are black, i.e. *Reach* $\mathsf{E} \subseteq$ *Blacks* M. Since it holds before *Append*, all white nodes are garbage right before the appending module starts. This is almost the safety property we wish to prove. The algorithm must ensure that only garbage nodes are collected during the appending phase. We shall show later when describing the *Append* module that if a white node is garbage before *Append*, then it remains so until *Append* makes it reachable.

The key parts of the invariant are the second and third lines. The second line guarantees that after any execution of the body the cardinalities of obc and bc are a lower and upper bound, respectively, of the number of black nodes after *Propagate-Black*. It is clear that obc is a lower bound, because black nodes stay black until the beginning of the appending phase. That bc is an upper bound would be difficult to prove without the auxiliary variable Ma since the mutator can blacken nodes while the collector executes *Count-Blacks*. The third line of the invariant guarantees that, if an execution of the body does not establish the safety property, then obc is a proper lower bound, which means that some white node was colored black during the execution of *Propagate-Black*. As we shall see, the *Propagate-Black* and *Count-Blacks* modules have very clear tasks: *Propagate-Black* establishes the third line, while *Count-Blacks* establishes the second.

Let us now see that the conjunction of the invariant and the negation of the guard $\mathsf{obc} \neq \mathsf{bc}$ imply the safety condition. If $\mathsf{obc} = \mathsf{bc}$, then the upper and lower bound coincide, and so obc cannot be a proper lower bound. Hence, no white node was colored black during *Propagate-Black*, and we obtain *Safe* (M, E).

We prove the derivability of the collector's proof outline. Since it is an infinite loop, the postcondition is the empty set of states, i.e. the set of states that satisfy the predicate *False*:

lemma *Collector*: $\vdash$ *Collector* $\{\!|$ *False* $|\!\}$

Blackening Roots

In this module a loop visits all roots and colors them black. The corresponding annotated command is:

defs *Blacken-Roots-def*:
 Blacken-Roots $\equiv$

```
{| Proper |}
ind := 0;;
{| Proper ∧ ind = 0 |}
while ind < length M
  inv {|Proper ∧ (∀ i < ind. i ∈ Roots ⟶ M!i = Black) ∧ ind ≤ length M|}
do {| Proper ∧ (∀ i < ind. i ∈ Roots ⟶ M!i = Black) ∧ ind < length M |}
 if ind ∈ Roots then
 {| Proper ∧ (∀ i < ind. i ∈ Roots ⟶ M!i = Black) ∧ ind < length M
    ∧ ind ∈ Roots |}
  M := M [ind := Black] fi;;
 {|Proper ∧ (∀ i < ind + 1. i ∈ Roots ⟶ M!i = Black) ∧ ind < length M|}
  ind := ind + 1
od
```

This module establishes in the postcondition that the set of roots is a subset of the set of black nodes. Its derivability in the *ann-hoare* system is easy to prove.

lemma *Blacken-Roots*: ⊢ *Blacken-Roots* {| Proper ∧ *Roots* ⊆ *Blacks* M |}

Propagation of the Coloring

During this phase, the collector visits the edges in a given order, coloring the target whenever the source is *Black*. This phase establishes the third line of the invariant.

The predicate *PBInv* contains the main idea of the proof. We declare it now but explain its meaning below:

consts
  PBInv :: *gar-coll-state* ⇒ *nat* ⇒ *bool*

We first explain an easier version of the *Propagate-Black* module. It will be later modified to obtain a finer degree of interleaving.

constdefs
  Propagate-Black-aux :: *gar-coll-state ann-com*
  Propagate-Black-aux ≡

```
{| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M |}
ind := 0;;
{| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
 ∧ ind = 0 |}
```

```
**while** $\mathsf{ind} < \mathit{length}\ \mathsf{E}$
 **inv** $\{\!|\ \mathsf{Proper} \wedge \mathit{Roots} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{obc} \subseteq \mathit{Blacks}\ \mathsf{M}$
        $\wedge\ \mathsf{bc} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{PBInv}\ \mathsf{ind} \wedge \mathsf{ind} \leq \mathit{length}\ \mathsf{E}\ |\!\}$
**do** $\{\!|\ \mathsf{Proper} \wedge \mathit{Roots} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{obc} \subseteq \mathit{Blacks}\ \mathsf{M}$
      $\wedge\ \mathsf{bc} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{PBInv}\ \mathsf{ind} \wedge \mathsf{ind} < \mathit{length}\ \mathsf{E}\ |\!\}$
 **if** $\mathsf{M}!\mathit{fst}\ (\mathsf{E}!\mathsf{ind}) = \mathit{Black}$ **then**
 $\{\!|\ \mathsf{Proper} \wedge \mathit{Roots} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{obc} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{bc} \subseteq \mathit{Blacks}\ \mathsf{M}$
   $\wedge\ \mathsf{PBInv}\ \mathsf{ind} \wedge \mathsf{ind} < \mathit{length}\ \mathsf{E} \wedge \mathsf{M}!\mathit{fst}\ (\mathsf{E}!\mathsf{ind}) = \mathit{Black}\ |\!\}$
  $\mathsf{M} := \mathsf{M}\ [\mathit{snd}\ (\mathsf{E}!\mathsf{ind}) := \mathit{Black}];;$
 $\{\!|\ \mathsf{Proper} \wedge \mathit{Roots} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{obc} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{bc} \subseteq \mathit{Blacks}\ \mathsf{M}$
   $\wedge\ \mathsf{PBInv}\ (\mathsf{ind} + 1) \wedge \mathsf{ind} < \mathit{length}\ \mathsf{E}\ |\!\}$
  $\mathsf{ind} := \mathsf{ind} + 1$
 **fi**
**od**
```

If the collector is executed in isolation it suffices to define PBInv as

$$\lambda ind.\ \mathsf{obc} \subset \mathit{Blacks}\ \mathsf{M} \vee (\forall\, i < ind.\ \neg \mathit{BtoW}\ (\mathsf{E}!i,\ \mathsf{M})).$$

Upon termination, i.e. when $\mathsf{ind} = \mathit{length}\ \mathsf{E}$ we would obtain

$$\mathsf{obc} \subset \mathit{Blacks}\ \mathsf{M} \vee (\forall\, i < \mathit{length}\ \mathsf{E}.\ \neg \mathit{BtoW}\ (\mathsf{E}!i,\ \mathsf{M}))$$

in the postcondition. When all roots are black, the following lemma holds:

lemma *Graph2*:
$[\![\mathit{Roots} \subseteq \mathit{Blacks}\ m;\ \mathit{Proper\text{-}Edges}\ (m,\ e);\ \forall\, i < \mathit{length}\ e.\ \neg \mathit{BtoW}\ (e!i,\ m)\ ]\!]$
$\Longrightarrow \mathit{Reach}\ e \subseteq \mathit{Blacks}\ m$

Hence, upon termination we obtain $\mathsf{obc} \subset \mathit{Blacks}\ \mathsf{M} \vee \mathit{Safe}\ (\mathsf{M},\ \mathsf{E})$, which is the postcondition of the *Propagate-Black* phase in the proof outline of the collector.

However, it is easy to see that this definition of *BPInv* is not invariant under the first action of the mutator: an edge that has already been visited by the collector could be redirected by the mutator to a white target. This is depicted in figure 3.4. In this example the collector is interrupted by the mutator when it reaches edge 3 ($\mathsf{ind} = 3$) but before coloring. The mutator then redirects the already visited edge 2 to node 3. Thus, there is a visited edge that satisfies the black-to-white predicate, falsifying the invariant.

Fortunately, we can find a weaker predicate that is able to establish the postcondition while remaining invariant under the actions of the mutator.

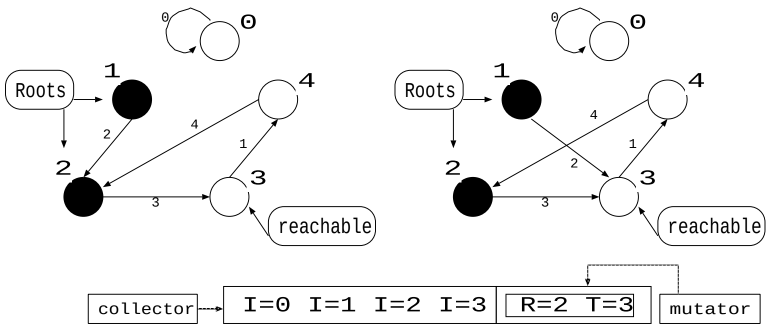

Figure 3.4: Interference with the mutator.

The following definition of *BPInv* is an adaptation of the one proposed in [van de Snepscheut, 1987]:

defs *PBInv-def*:
$PBInv \equiv \ll \lambda ind.\ \mathsf{obc} < Blacks\ \mathsf{M} \lor (\forall\, i < ind.\ \neg BtoW\ (\mathsf{E}!i,\ \mathsf{M}) \lor$
$(\neg \mathsf{z} \land i = R \land (snd\ (\mathsf{E}!R)) = T \land$
$(\exists\, r.\ ind \leq r \land r < length\ \mathsf{E} \land BtoW\ (\mathsf{E}!r,\ \mathsf{M})))) \gg$

Intuitively, its invariance is proved as follows.

If either the collector or the mutator blacken some white node then, after execution of the body, the predicate $\mathsf{obc} \subset Blacks\ \mathsf{M}$ holds. Otherwise, i.e. no coloring occurs, there are two situations:

1. All edges visited by the collector point to a *Black* node, i.e. $\forall\, i < \mathsf{ind}.\ \neg BtoW\ (\mathsf{E}!i,\ \mathsf{M})$ holds.

2. Some visited edge points to a white node because the mutator has redirected it. Then this edge has target node T. Ben-Ari observes that in this situation there must be another *BtoW* edge among those that have not yet been visited by the collector. This holds because the new target T is reachable by assumption, thus, there exists a path to T from some root. Since all roots are black, some edge along this path must be a *BtoW* edge.

Observe that upon termination of the loop this last clause cannot hold because the counter ind reaches the value of *length* E. Consequently, this invariant establishes $\mathsf{obc} \subset Blacks\ \mathsf{M} \lor Safe\ (\mathsf{M},\ \mathsf{E})$ upon termination and is interference free under the mutator's actions. We prove the derivability of the corresponding triple:

lemma *Propagate-Black-aux*:
$\vdash$ *Propagate-Black-aux*
$\{\!\mid$ $\mathsf{Proper} \wedge \mathit{Roots} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{obc} \subseteq \mathit{Blacks}\ \mathsf{M} \wedge \mathsf{bc} \subseteq \mathit{Blacks}\ \mathsf{M}$
$\wedge (\mathsf{obc} \subset \mathit{Blacks}\ \mathsf{M} \vee \mathit{Safe}\ (\mathsf{M}, \mathsf{E}))$ $\mid\!\}$

The assignment $\mathsf{M} := \mathsf{M}\ [\mathit{snd}\ (\mathsf{E}!\mathsf{ind}) := \mathit{Black}]$ contains two references to shared variables (M and E), which leads to implementation problems. Fortunately, the loop body can be replaced by

```
if M! fst (E!ind) = Black
then k := snd (E!ind);; ⟨M := M [k := Black],, ind := ind + 1⟩
else ⟨if M! fst (E!ind) ≠ Black then ind := ind + 1 fi⟩
```

where at most one shared variable is accessed in each atomic action.

This introduces a new point of interference with the mutator. After this modification the program remains correct although the precondition of the atomic region $\langle \mathsf{M} := \mathsf{M}\ [\mathsf{k} := \mathit{Black}],,\ \mathsf{ind} := \mathsf{ind} + 1\rangle$ is non-trivial. It includes the following predicate, proposed in [van de Snepscheut, 1987]:

constdefs
Auxk :: *gar-coll-state* $\Rightarrow$ *bool*
$\mathit{Auxk} \equiv \ll \mathsf{k} < \mathit{length}\ \mathsf{M} \wedge (\mathsf{M}!\mathsf{k} \neq \mathit{Black} \vee \neg \mathit{BtoW}\ (\mathsf{E}!\mathsf{ind}, \mathsf{M})$
$\vee\ \mathsf{obc} < \mathit{Blacks}\ \mathsf{M} \vee (\neg \mathsf{z} \wedge \mathsf{ind} = R \wedge \mathit{snd}\ (\mathsf{E}!R) = T$
$\wedge\ (\exists\, r.\ \mathsf{ind} < r \wedge r < \mathit{length}\ \mathsf{E} \wedge \mathit{BtoW}\ (\mathsf{E}!r, \mathsf{M})))) \gg$

Van de Snepscheut leaves its verification as an exercise that we carried out successfully.

defs
Propagate-Black-def:
Propagate-Black $\equiv$

```
{| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M |}
ind := 0;;
{| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M
  ∧ bc ⊆ Blacks M ∧ ind = 0 |}
while ind < length E
  inv {| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M
         ∧ bc ⊆ Blacks M ∧ PBInv ind ∧ ind ≤ length E |}
do {| Proper ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M
      ∧ bc ⊆ Blacks M ∧ PBInv ind ∧ ind < length E |}
 if M! fst (E!ind) = Black
 then
   ⦃ Proper ∧ $Roots$ ⊆ $Blacks$ M ∧ obc ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
     ∧ PBInv ind ∧ ind < $length$ E ∧ M!$fst$ (E!ind) = $Black$ ⦄
   k := $snd$ (E!ind);;
   ⦃ Proper ∧ $Roots$ ⊆ $Blacks$ M ∧ obc ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
     ∧ PBInv ind ∧ ind < $length$ E ∧ M!$fst$ (E!ind) = $Black$ ∧ Auxk ⦄
   ⟨M := M [k := $Black$],, ind := ind + 1⟩
 else
   ⦃ Proper ∧ $Roots$ ⊆ $Blacks$ M ∧ obc ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
     ∧ PBInv ind ∧ ind < $length$ E ⦄
   ⟨if M! $fst$ (E!ind) ≠ $Black$ then ind := ind + 1 fi⟩
 fi
od
```

The evaluation of the condition M! *fst* (E!ind) ≠ *Black* and the incrementing of ind must be done atomically. If we let a point of interference in between, the mutator could blacken the node, which would falsify the assertion.

lemma *Propagate-Black*:
 ⊢ *Propagate-Black*
⦃ Proper ∧ *Roots* ⊆ *Blacks* M ∧ obc ⊆ *Blacks* M ∧ bc ⊆ *Blacks* M
 ∧ (obc ⊂ *Blacks* M ∨ *Safe* (M, E)) ⦄

Counting Black Nodes

This phase finally re-establishes the invariant of the collector's outermost loop. The computed set bc must contain all nodes which were black *upon termination of Propagate-Black* or, since Ma records precisely this set, the *Count-Blacks* phase must ensure that *Blacks* Ma ⊆ bc holds.

The invariant contains the predicate

constdefs
  CountInv :: *gar-coll-state* ⇒ *nat* ⇒ *bool*
  CountInv ≡ ≪ $\lambda ind.\ \{i.\ i < ind \wedge$ Ma!$i = Black\} \subseteq$ bc ≫

which upon termination establishes *Blacks* Ma ⊆ bc. The corresponding annotated command is:

defs
  Count-Blacks-def:
  Count-Blacks ≡
⦃ Proper ∧ *Roots* ⊆ *Blacks* M ∧ *length* Ma = *length* M

```
  ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M
  ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) ∧ bc = {} ⦄
ind := 0;;
⦃ Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
  ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M
  ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) ∧ bc = {} ∧ ind = 0 ⦄
while ind < length M
  inv ⦃ Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
        ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M
        ∧ CountInv ind
        ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) ∧ ind ≤ length M ⦄
do ⦃ Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
     ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M
     ∧ CountInv ind
     ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) ∧ ind < length M ⦄
   if M!ind = Black
    then ⦃ Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
           ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M
           ∧ CountInv ind ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E))
           ∧ ind < length M ∧ M!ind = Black ⦄
    bc := insert ind bc
   fi;;
   ⦃ Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
    ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ Blacks M ∧ bc ⊆ Blacks M
    ∧ CountInv (ind + 1)
    ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) ∧ ind < length M ⦄
   ind := ind + 1
od
```

The mutator cannot access the auxiliary variable Ma. Thus, the set of black nodes of Ma remains invariant under the mutator's blackening action so that these annotations are invariant against the mutator's actions.

The postcondition is exactly the outermost invariant of the collector's loop, which must hold at the beginning and at the end of the loop's body.

```
lemma Count-Blacks:
 ⊢ Count-Blacks
 ⦃ Proper ∧ Roots ⊆ Blacks M ∧ length Ma = length M
  ∧ obc ⊆ Blacks Ma ∧ Blacks Ma ⊆ bc ∧ bc ⊆ Blacks M
  ∧ (obc ⊂ Blacks Ma ∨ Safe (M, E)) ⦄
```

With all reachable nodes marked we can proceed to the appending phase where all unmarked nodes are appended to the free list.

Appending to the Free List

Following our predecessors, the operation of appending a garbage node ind to the free list, i.e. making ind reachable, is modeled abstractly by the function:

consts $\textit{Append-to-free} :: \textit{nat} \times \textit{edges} \Rightarrow \textit{edges}$

satisfying the following axioms:

axioms

```
  $Append\text{-}to\text{-}free_0$: $length\ (Append\text{-}to\text{-}free\ (n,\ e)) = length\ e$
  $Append\text{-}to\text{-}free_1$: $Proper\text{-}Edges\ (m,\ e)$
                      $\Longrightarrow Proper\text{-}Edges\ (m,\ Append\text{-}to\text{-}free\ (n,\ e))$
  $Append\text{-}to\text{-}free_2$: $n \notin Reach\ e$
          $\Longrightarrow n' \in Reach\ (Append\text{-}to\text{-}free\ (n,\ e)) = (n' = n \lor n' \in Reach\ e)$
```

In the annotated code, AppendInv ind states that all white nodes with index ind or larger are garbage, i.e. the safety property is maintained throughout the appending loop.

constdefs

```
  $AppendInv :: gar\text{-}coll\text{-}state \Rightarrow nat \Rightarrow bool$
  $AppendInv \equiv \ll\lambda ind.\ \forall i{<}length\ \mathsf{M}.\ ind \le i \longrightarrow i \in Reach\ \mathsf{E} \longrightarrow \mathsf{M}!i = Black\gg$
```

defs

```
  $Append\text{-}def$:
  $Append \equiv$
 $\{\!|$ Proper $\land$ $Roots \subseteq Blacks$ M $\land$ $Safe$ (M, E) $|\!\}$
 ind := 0;;
 $\{\!|$ Proper $\land$ $Roots \subseteq Blacks$ M $\land$ $Safe$ (M, E) $\land$ ind = 0 $|\!\}$
 while ind $<$ $length$ M
   inv $\{\!|$ Proper $\land$ AppendInv ind $\land$ ind $\le$ $length$ M $|\!\}$
 do $\{\!|$ Proper $\land$ AppendInv ind $\land$ ind $<$ $length$ M $|\!\}$
    if M!ind = $Black$ then
     $\{\!|$ Proper $\land$ AppendInv ind $\land$ ind $<$ $length$ M $\land$ M!ind = $Black$ $|\!\}$
       M := M [ind := $White$]
     else
```

```
      {| Proper ∧ AppendInv ind ∧ ind < length M ∧ ind ∉ Reach E |}
      E := Append-to-free (ind, E)
   fi;;
   {| Proper ∧ AppendInv (ind + 1) ∧ ind < length M |}
   ind := ind + 1
od
```

The precondition of the assignment to E guarantees that only garbage nodes are collected.

3.3.3 Interference Freedom

The proof outline for the mutator has a total of 5 assertions and 2 atomic actions. The proof outline of the collector has 36 assertions and 20 atomic actions. Hence, the number of interference freedom tests that have to be checked is 172. Obviously many of them are trivial, but in many cases what seemed to be a perfect proof outline revealed a bug only after attempting the proof with the theorem prover.

We carry out part of the interference freeness tests using the modules used to structure the code of the collector and mutator. We prove lemmas about the invariance of the assertions in each module of the collector against the atomic actions in each module of the mutator, and vice versa. These are in total 16 lemmas of the form:

lemma *interfree-Blacken-Roots--Redirect-Edge*:
interfree-aux (*Some Blacken-Roots*, {}, *Some Redirect-Edge*)
lemma *interfree-Redirect-Edge--Blacken-Roots*:
interfree-aux (*Some Redirect-Edge*, {}, *Some Blacken-Roots*)

The verification conditions that result from the interference-freedom tests represented by these lemmas can be automatically generated. First, the definitions of the modules are unfolded, and then we apply a special tactic called *interfree-aux* which is a “subtactic” of the one used for full parallel programs (see appendix A.3).

To prove several verification conditions that result from the interference freedom tests, we need some auxiliary lemmas about graphs. The first one states that the set of reachable nodes is not increased by the first action of the mutator:

lemma *Graph*3:
$[\![\ t \in \mathit{Reach}\ e;\ r < \mathit{length}\ e\ ]\!] \Longrightarrow \mathit{Reach}\ (e\ [r := (\mathit{fst}\ (e!r),\ t)]) \subseteq \mathit{Reach}\ e$

In the proof of *interfree-Propagate-Black--Redirect-Edge* we need the following lemma:

lemma *Graph*4:
$[\![t \in \mathit{Reach}\ e;\ \mathit{Roots} \subseteq \mathit{Blacks}\ m;\ \mathit{index} \le \mathit{length}\ e;\ t < \mathit{length}\ m;\ r < \mathit{length}\ e;$
$\quad \forall\, i < \mathit{index}.\ \neg \mathit{BtoW}\ (e!i,\ m);\ r < \mathit{index};\ m!\mathit{fst}\ (e!r) = \mathit{Black};\ m!t \neq \mathit{Black}]\!]$
$\Longrightarrow \exists\, r.\ \mathit{index} \le r \wedge r < \mathit{length}\ e \wedge \mathit{BtoW}\ (e\ [r := (\mathit{fst}\ (e!r),\ t)]!r,\ m)$

establishing that whenever a visited edge $r < \mathit{index}$ is redirected to a white target t then there exists a not visited edge in the modified list of edges such that it satisfies the predicate *BtoW* as well. A slight variation is needed to prove the invariance of the predicate *Auxk*, namely,

lemma *Graph*5:
$[\![\ t \in \mathit{Reach}\ e;\ \mathit{Roots} \subseteq \mathit{Blacks}\ m;\ \forall\, i < r.\ \neg \mathit{BtoW}\ (e!i,\ m);\ t < \mathit{length}\ m;$
$\quad r < \mathit{length}\ e;\ m!\mathit{fst}\ (e!r) = \mathit{Black};\ m!\mathit{snd}\ (e!r) = \mathit{Black};\ m!t \neq \mathit{Black}]\!]$
$\Longrightarrow \exists\, r'.\ r < r' \wedge r' < \mathit{length}\ e \wedge \mathit{BtoW}\ (e\ [r := (\mathit{fst}\ (e!r),\ t)]!r',\ m)$

Next, we prove the interference freedom of the collector against the mutator and vice versa:

lemma *interfree-Collector--Mutator*:
interfree-aux (*Some Collector*, {}, *Some Mutator*)
lemma *interfree-Mutator--Collector*:
interfree-aux (*Some Mutator*, {}, *Some Collector*)

Finally, we prove the derivability of the full program:

lemma *Gar-Coll*:
$\Vdash$ {| Proper $\wedge$ Mut-init $\wedge$ z |}
cobegin
  Collector {| *False* |} ‖ *Mutator* {| *False* |}
coend
{| *False* |}

The tactic *oghoare* is applied without unfolding the definitions of the modules. As a result, the derivability of the components is directly proven by the lemmas *Collector* and *Mutator*, and the interference freedom test consists only of two subgoals, which correspond exactly to the lemmas about interference shown above. By the soundness theorem the algorithm is correct in the sense of partial correctness. The validity of the postcondition is, however, not interesting since both the mutator and the collector are infinite cycles. The interesting property is the validity of the intermediate annotations. In particular, the precondition of the action which appends nodes to the free list ensures that these nodes are garbage.

3.4 The Multi-Mutator Case

If we allow the interaction with several mutators, new difficulties come into play. We consider a solution, first presented in [Jonker, 1992], where the collector proceeds to the appending phase only after $n+1$ consecutive executions of the *Propagate-Black* phase where the set of black nodes is not increased. Observe that, for one mutator, this algorithm checks *twice* whether obc = bc. [Jonker, 1992] also shows that n consecutive executions suffice, but we do not consider this version here.

The program consists of a fixed, finite and nonempty set of mutator processes and one collector process. The external syntax for parameterized programs is shown in the table 2.4.

3.4.1 The Mutators

A mutator can only redirect an edge when its target is a reachable node. Redirecting an edge may make its old target inaccessible. If several mutators are active, then one of them may select a reachable node T as a new target. Before the edge has been redirected, however, another mutator may render T inaccessible. To solve this problem, selecting the new target and redirecting the edge is modeled as a single atomic action.

Each mutator m selects an edge R_m and a target node T_m. As in the previous section each mutator uses an auxiliary variable Z_m, that indicates if it is pending before the blackening of a node. These three objects are put together as fields of a record:

record *mut* =
  Z :: *bool*
  R :: *nat*
  T :: *nat*

Isabelle's syntax for accessing the field Z of a variable Mut of type *mut* is Z Mut. Record update is written Mut $(\!| Z{:=}True |\!)$, meaning that the field Z of the record Mut is updated to the value *True*.

In the algorithm the variable Muts is a list of length n (the number of mutators) whose components are records of type *mut*. For example, to access the selected edge of mutator j we write R (Muts!j).

The variables of the program are the same as in the case for one mutator, except for the list Muts used by the mutator and two new variables, Qa and L, of the collector which we explain in the following sections:

```
record mul-gar-coll-state =
  M :: nodes
  E :: edges
  bc :: nat set
  obc :: nat set
  Ma :: nodes
  ind :: nat
  k :: nat
  Qa :: nat
  L :: nat
  Muts :: mut list
```

In the assertions of the mutator we use the following predicate:

```
constdefs
  Mul-mut-init :: mul-gar-coll-state ⇒ nat ⇒ bool
  Mul-mut-init ≡ « λn. n = length Muts ∧ (∀ i < n. R (Muts!i) < length E
                        ∧ T (Muts!i) < length M) »
```

indicating that the selected edges and targets are within the range of edges and nodes, respectively, and that the list of records Muts has an entry for each of the n mutators.

The modules of the mutator's code are functions of the number of mutators n and the particular mutator's index j with $0 \leq j < n$.

```
constdefs
  Mul-Redirect-Edge :: nat ⇒ nat ⇒ mul-gar-coll-state ann-com
  Mul-Redirect-Edge j n ≡
  {| Mul-mut-init n ∧ Z (Muts!j) |}
  ⟨if T (Muts!j) ∈ Reach E then
  E := E [R (Muts!j) := (fst (E!R (Muts!j)), T (Muts!j))] fi,,
  Muts := Muts [j := (Muts!j) (|Z := False|)]⟩

  Mul-Color-Target :: nat ⇒ nat ⇒ mul-gar-coll-state ann-com
```

```
$Mul\text{-}Color\text{-}Target\ j\ n \equiv$
$\{\!|$ Mul-mut-init $n \wedge \neg\ Z$ (Muts!$j$) $|\!\}$
$\langle$M := M $[T$ (Muts!$j$) := $Black]$,, Muts := Muts $[j$ := (Muts!$j$) $(\!|Z := True|\!)]\rangle$

$Mul\text{-}Mutator :: nat \Rightarrow nat \Rightarrow mul\text{-}gar\text{-}coll\text{-}state\ ann\text{-}com$
$Mul\text{-}Mutator\ j\ n \equiv$
$\{\!|$ Mul-mut-init $n \wedge Z$ (Muts!$j$) $|\!\}$
while $True$
  inv $\{\!|$ Mul-mut-init $n \wedge Z$ (Muts!$j$) $|\!\}$
do $Mul\text{-}Redirect\text{-}Edge\ j\ n$ ;;
    $Mul\text{-}Color\text{-}Target\ j\ n$
od
```

The annotations of the proof outline of the mutators are, like the instructions, parameterized by the number of mutators n and the index j. In chapter 5, we shall show that parameterized annotations for correct specifications of parameterized programs can always be found. We prove the derivability of the parameterized proof outline of a generic mutator:

lemma *Mul-Mutator*: $[\![0 \leq j;\ j < n]\!] \Longrightarrow \vdash Mul\text{-}Mutator\ j\ n\ \{\!|\ False\ |\!\}$

3.4.2 The Collector

In the case of one mutator, if an execution of the body does not establish the safety property, the reason is that some white node was colored black during the execution of *Propagate-Black*. When several mutators are present, there may be other reasons. To describe them we need a new value which represents the number of mutators that are *queueing* to blacken a white node. This value is computed by the function *Queue*, which returns the length of the list that results from filtering the queueing mutators from the list of mutator variables:

constdefs
$Queue :: mul\text{-}gar\text{-}coll\text{-}state \Rightarrow nat$
$Queue \equiv \ll length\ (filter\ (\lambda i.\ \neg\ Z\ i \wedge \mathsf{M}!(T\ i) \neq Black)\ \mathsf{Muts}) \gg$

The auxiliary variable Qa “records” this value upon termination of the *Propagate-Black* phase. The definition of the predicate *Mul-Proper* requires, besides proper nodes and proper edges, that the length of the variable Muts be the number of mutators n:

constdefs
  Mul-Proper :: *mul-gar-coll-state* ⇒ *nat* ⇒ *bool*
  Mul-Proper ≡ ≪ λ*n*. *Proper-Roots* M ∧ *Proper-Edges* (M, E)
                ∧ *Proper-M-init* M ∧ n = *length* Muts ≫

We declare the modules used for the collector:

consts
  Mul-Blacken-Roots :: *nat* ⇒ *mul-gar-coll-state ann-com*
  Mul-Propagate-Black :: *nat* ⇒ *mul-gar-coll-state ann-com*
  Mul-Count-Blacks :: *nat* ⇒ *mul-gar-coll-state ann-com*
  Mul-Append :: *nat* ⇒ *mul-gar-coll-state ann-com*

The variable L is a counter that keeps track of how many consecutive times the values of obc and bc coincide. When it reaches the value $n + 1$ the conditions satisfy the safety requirement and the collector proceeds to collect the unmarked nodes. The proof outline of the collector is:

constdefs

```
  Mul-Collector :: nat ⇒ mul-gar-coll-state ann-com
  Mul-Collector n ≡
 {| Mul-Proper n |}
 while True inv {| Mul-Proper n |}
 do
 Mul-Blacken-Roots n ;;
 {| Mul-Proper n ∧ Roots ⊆ Blacks M |} obc := {};;
 {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc = {} |} bc := Roots;;
 {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc = {} ∧ bc = Roots |} L := 0;;
 {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc = {} ∧ bc = Roots ∧ L = 0 |}
 while L < n + 1
   inv {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ bc ⊆ Blacks M ∧
          (Safe (M, E) ∨ (L ≤ Queue ∨ bc ⊂ Blacks M) ∧ L < n + 1) |}
 do
   {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ bc ⊆ Blacks M
    ∧ (Safe (M, E) ∨ L ≤ Queue ∨ bc ⊂ Blacks M) |}
   obc := bc;;
   Mul-Propagate-Black n;;
   {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
    ∧ (Safe (M, E) ∨ obc ⊂ Blacks M ∨ L < Queue
    ∧ (L ≤ Queue ∨ obc ⊂ Blacks M)) |}
  bc := {};;
  ⦃ Mul-Proper $n$ ∧ $Roots$ ⊆ $Blacks$ M ∧ obc ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
   ∧ ($Safe$ (M, E) ∨ obc ⊂ $Blacks$ M ∨ L < Queue
   ∧ (L ≤ Queue ∨ obc ⊂ $Blacks$ M)) ∧ bc = {} ⦄
  ⟨Ma := M,, Qa := Queue⟩;;
  $Mul\text{-}Count\text{-}Blacks\ n$;;
  ⦃ Mul-Proper $n$ ∧ $Roots$ ⊆ $Blacks$ M ∧ $length$ Ma = $length$ M
   ∧ obc ⊆ $Blacks$ Ma ∧ $Blacks$ Ma ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
   ∧ $Blacks$ Ma ⊆ bc ∧ ($Safe$ (M, E) ∨ obc ⊂ $Blacks$ Ma ∨
      L < Qa ∧ (Qa ≤ Queue ∨ obc ⊂ $Blacks$ M)) ∧ Qa < $n + 1$ ⦄
 **if** obc = bc
  **then**
   ⦃ Mul-Proper $n$ ∧ $Roots$ ⊆ $Blacks$ M ∧ $length$ Ma = $length$ M
    ∧ obc ⊆ $Blacks$ Ma ∧ $Blacks$ Ma ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
    ∧ $Blacks$ Ma ⊆ bc ∧ ($Safe$ (M, E) ∨ obc ⊂ $Blacks$ Ma ∨
       L < Qa ∧ (Qa ≤ Queue ∨ obc ⊂ $Blacks$ M)) ∧ Qa < $n + 1$
    ∧ obc = bc ⦄
   L := L + 1
  **else**
   ⦃ Mul-Proper $n$ ∧ $Roots$ ⊆ $Blacks$ M ∧ $length$ Ma = $length$ M
    ∧ obc ⊆ $Blacks$ Ma ∧ $Blacks$ Ma ⊆ $Blacks$ M ∧ bc ⊆ $Blacks$ M
    ∧ $Blacks$ Ma ⊆ bc ∧ ($Safe$ (M, E) ∨ obc ⊂ $Blacks$ Ma ∨
       L < Qa ∧ (Qa ≤ Queue ∨ obc ⊂ $Blacks$ M)) ∧ Qa < $n + 1$
    ∧ obc ≠ bc ⦄
   L := 0
 **fi**
**od**;;
$Mul\text{-}Append\ n$
**od**
```

The invariant of the one-mutator case must be compared with the precondition of the **if**−**then**−**else** instruction, because both correspond to the assertion established by the phase that counts the black nodes. The assertion *Safe* (M, E) ∨ obc ⊂ *Blacks* Ma has been weakened with a new disjunct, corresponding to the new situation which can prevent *Safe* (M, E) from holding. The new disjunct corresponds to the case in which at least one mutator joins the queue during the *Mul-Propagate-Black* phase, i.e. L < Qa. In this case it is also necessary to distinguish whether some mutator leaves the queue, i.e. colors its white target (obc ⊂ *Blacks* M), or none leaves the queue, i.e. the queue is not decreased (Qa ≤ Queue). Intuitively, after $n+1$

non-blackening *Mul-Propagate-Black* iterations, the property *Safe* (M, E) must hold, since the number of queueing mutators cannot exceed n.

The codes of the modules are the same as in §3.3 except for the annotations in the *Mul-Propagate-Black* and *Count-Blacks* phases, which have to be adapted to the new invariant.

We just show the *Mul-Propagate-Black* phase.

```
constdefs
  Mul-PBInv :: mul-gar-coll-state ⇒ bool
  Mul-PBInv ≡ « Safe (M, E) ∨ obc ⊂ Blacks M ∨ L < Queue
                ∨ (∀ i < ind. ¬BtoW (E!i, M)) ∧ L ≤ Queue »

  Mul-Auxk :: mul-gar-coll-state ⇒ bool
  Mul-Auxk ≡ « L < Queue ∨ M!k ≠ Black ∨ ¬BtoW (E!ind, M)
                ∨ obc ⊂ Blacks M »


defs
  Mul-Propagate-Black-def:
  Mul-Propagate-Black n ≡
 {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
   ∧ (Safe (M, E) ∨ L ≤ Queue ∨ obc ⊂ Blacks M) |}
 ind := 0;;
 {| Mul-Proper n ∧ Roots ⊆ Blacks M
   ∧ obc ⊆ Blacks M ∧ Blacks M ⊆ Blacks M ∧ bc ⊆ Blacks M
   ∧ (Safe (M, E) ∨ L ≤ Queue ∨ obc ⊂ Blacks M) ∧ ind = 0 |}
 while ind < length E
  inv {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
         ∧ Mul-PBInv ∧ ind ≤ length E |}
 do {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
        ∧ Mul-PBInv ∧ ind < length E |}
  if M!fst (E!ind) = Black
  then
   {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
     ∧ Mul-PBInv ∧ M!fst (E!ind) = Black ∧ ind < length E |}
   k := snd (E!ind);;
   {| Mul-Proper n ∧ Roots ⊆ Blacks M ∧ obc ⊆ Blacks M ∧ bc ⊆ Blacks M
     ∧ (Safe (M, E) ∨ obc ⊂ Blacks M ∨ L < Queue ∨
       (∀ i < ind. ¬BtoW (E!i, M)) ∧ L ≤ Queue ∧ Mul-Auxk )
    ∧ k < length M ∧ M!fst (E!ind) = Black ∧ ind < length E |}
   ⟨M := M [k := Black],, ind := ind + 1⟩
 else
  $\{\!|$ Mul-Proper $n \wedge Roots \subseteq Blacks$ M $\wedge$ obc $\subseteq Blacks$ M $\wedge$ bc $\subseteq Blacks$ M
     $\wedge$ Mul-PBInv $\wedge$ ind $< length$ E $|\!\}$
  $\langle$if M!$fst$ (E!ind) $\neq Black$ then ind := ind + 1 fi$\rangle$
 fi
od
```

If we expand the predicate Mul-PBInv in the invariant we obtain

$$Safe\ (\mathsf{M}, \mathsf{E}) \vee \mathsf{obc} \subset Blacks\ \mathsf{M} \vee \mathsf{L} < \mathsf{Queue}$$
$$\vee\ (\forall\, i < \mathsf{ind}.\ \neg BtoW\ (\mathsf{E}!i, \mathsf{M})) \wedge \mathsf{L} \leq \mathsf{Queue}.$$

Any coloring establishes obc $\subset$ *Blacks* M. (Observe that only coloring can make the queue shorter.) If no coloring occurs, then, either all the visited edges point to a black node, or some mutator has redirected an edge to a white source, but has not yet colored the target, which amounts to saying that the queue grows, i.e. L $<$ Queue.

The next lemma proves derivability of the collector's proof outline in the system:

lemma *Mul-Collector*: $\vdash$ *Mul-Collector n* $\{\!|$ *False* $|\!\}$

3.4.3 Interference Freedom

The collector has a total of 40 assertions and 21 atomic actions. One mutator has 5 assertions and 2 atomic actions. For the interference freedom test it suffices to consider the following combinations:

1. Invariance of the assertions in the collector against the actions of a generic mutator with some index j such that $0 \leq j < n$, and vice versa.

2. Invariance of the assertions of one mutator j against the actions of another mutator i such that $i \neq j$.

This results in a total of 195 interference freedom proofs. Like for the one-mutator case we perform combinations of the modules in separate lemmas that are then applied directly to the verification of the parallel composition. The interference freedom among the mutators is proven in the following lemma:

lemma *Mul-interfree-Mutator--Mutator*: $[\![i < n; j < n; i \neq j]\!] \Longrightarrow$
  interfree-aux (*Some* (*Mul-Mutator i n*), {}, *Some* (*Mul-Mutator j n*))

The derivability of the parallel composition of the mutators, i.e. the lemma

lemma *Mul-Parameterized-Mutators*: $0 < n \Longrightarrow$
$\Vdash$ $\{\!|$ Mul-mut-init $n \wedge (\forall\, i < n.\ Z$ (Muts!i)) $|\!\}$
cobegin
  scheme $[0 \leq j < n]$ *Mul-Mutator* j n $\{\!|$ *False* $|\!\}$
coend
$\{\!|$ *False* $|\!\}$

is proven by applying the tactic *oghoare* without unfolding the definition of *Mul-Mutator*. The tactic generates four subgoals. Two of them correspond to the two verification conditions about the logical implications of the overall precondition and postcondition and those of the components. A third subgoal stating the derivability of one generic mutator (proven by the lemma *Mul-Mutator*), and the forth subgoal stating the interference between two generic mutators (proven by the lemma *Mul-interfree-Mutator--Mutator*). Notice that there is no need for induction or any other proof method; parameterized programs are directly handled by the proof rules of the system.

The rest of the interference freedom tests are similar to those of the one-mutator case:

lemma *Mul-interfree-Collector--Mutator*: $j < n \Longrightarrow$
  interfree-aux (*Some* (*Mul-Collector* n), {}, *Some* (*Mul-Mutator* j n))
lemma *Mul-interfree-Mutator--Collector*: $j < n \Longrightarrow$
  interfree-aux (*Some* (*Mul-Mutator* j n), {}, *Some* (*Mul-Collector* n))

Finally, we prove the derivability in the Owicki-Gries system of the full program:

lemma *Mul-Gar-Coll*:
$\Vdash$ $\{\!|$ Mul-Proper $n \wedge$ Mul-mut-init $n \wedge (\forall\, i < n.\ Z$ (Muts!i)) $|\!\}$
cobegin
 Mul-Collector n $\{\!|$ *False* $|\!\}$
 $\|$
 scheme $[0 \leq j < n]$ *Mul-Mutator* j n $\{\!|$ *False* $|\!\}$
coend
$\{\!|$ *False* $|\!\}$

3.5 Conclusions and Related Work

We have provided mechanically checked Owicki-Gries proofs for two garbage collection algorithms. 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 were automatically 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.

We do not know of any complete Owicki-Gries proof for any of the two algorithms. In his proof of Ben-Ari's algorithm, [van de Snepscheut, 1987] mixes the Owicki-Gries method with ad-hoc reasoning; in particular, he does not provide an invariant for the outermost loop, implicitly claiming that doing so would be complicated. However, the invariant turns out to be simple (3 clauses), and has a clear intuitive interpretation.

For the n-mutators algorithm, [Jonker, 1992] argues that

> A proof according to the Owicki-Gries theory would require the introduction of a satisfactory number of ghost variables. In an earlier version of this paper the invariant we constructed was rather unwieldy and the proof of invariance almost unreadable.

However, our proof only uses two auxiliary variables (Ma and Qa), plus a trivial auxiliary variable for each mutator. Jonker considers in his paper several variations of the algorithm. We believe that Owicki-Gries proofs for these variations should be possible to obtain from the proof presented here with reasonable effort.

We know of two other mechanized proofs of Ben-Ari's algorithm, carried out using the Boyer-Moore theorem prover [Russinoff, 1994] and the PVS theorem prover [Havelund, 1996, Havelund and Shankar, 1997]. A main advantage of our approach is the closeness to the original program text, which simplifies the interaction with the prover. Annotated programs are fairly readable by humans, and they are also directly accepted as input by Isabelle. In other approaches the program must first be translated into a different language (e.g. LISP in [Russinoff, 1994]).

Another aspect of our formalization is that we only had to prove 13 lemmas (7 of them trivial) about graph functions, whereas 100 lemmas were required in [Russinoff, 1994], and about 55 in [Havelund, 1996, Havelund and Shankar, 1997]. The reason for this is probably that many trivial lemmas about sets or lists could be automatically proven using Isabelle's built-in tactics (rewriting, classical reasoning, etc.) and Isabelle's standard libraries. The proof effort, however, took two months for the one-mutator algorithm (similar to our predecessors) and another two months for the n-mutator case.

Most of the time was consumed in finding and improving the annotations.

A disadvantage of the Owicki-Gries method (in its classical version) is that it can only be applied to safety properties. A liveness property, namely *every garbage node is eventually collected*, is also very important for garbage collection algorithms. Mechanical proofs of this liveness property are found in [Russinoff, 1994] for Ben Ari's algorithm and in [Jackson, 1998], where the safety and liveness property of a predecessor of Ben Ari's algorithm [Dijkstra *et al.*, 1978] is proven using PVS.

None of our two algorithms has been proven correct using fully automatic methods. In [Bruns, 1997], there is a proof of Ben Ari's algorithm for 1 mutator and 4 memory cells. In [Das *et al.*, 1999], a predecessor of Ben Ari's algorithm is proved correct using automatic tools for generating and proving invariants. The key invariants, however, require intelligent input from the user. The paper suggests using predicate abstraction for checking or strengthening invariants in a larger verification effort involving interactive theorem provers, which is a promising idea.

Our overall conclusion is that the application of a theorem prover greatly enhances the applicability of the Owicki-Gries method. The closeness to the original program is preserved, and the large number of routine proofs is considerably automated.

Chapter 4

The Rely-Guarantee Method in Isabelle/HOL

This chapter presents the formalization of the rely-guarantee method for correctness proofs of parallel imperative programs with shared variables in the theorem prover Isabelle/HOL. This method was first proposed by Jones [Jones, 1981, Jones, 1983] and can be seen as the compositional version of Owicki-Gries. We closely follow the presentation in [Xu *et al.*, 1997], where a sound and complete version of the system is presented in a conventional pencil and paper style. However, some aspects of our formalization differ from this model. The reasons for these modifications will be addressed as we encounter them.

The rely-guarantee system provides compositional proof rules for the verification of parallel programs. This is accomplished by enriching the specification of each component with conditions concerning the interaction with the environment. A rely-guarantee specification defines four sets: the sets of initial and final states (pre and postcondition) and the sets characterizing the effect of actions performed by the environment or by the component itself (rely and guarantee conditions).

It is important to observe the strong connection to Owicki-Gries. In the latter, programs were annotated at every point of interference and the verification process required proving interference freedom of the annotations of each component against the atomic actions of the other components. The idea of the rely-guarantee method is to record the interference information in the specification of each component. This information consists of the rely condition stating what the component expects from the environment, and the guarantee condition, stating the effect of the component itself on the environment. By using the information given by the rely and guarantee conditions of each component, the verification of a parallel program can be carried out in a compositional way.

The compositionality of this method has two major advantages over the non-compositionality of Owicki-Gries. First of all, it drastically reduces the complexity of the verification process. The number of correctness proofs in the rely-guarantee formalism increases only linearly with the number of parallel components, while exponentially in the Owicki-Gries method. Secondly, it allows the verification of so-called *open systems*, i.e. systems where a specified margin of interaction from an arbitrary environment is permitted. Then, new components, whose actions respect this margin, can be added a posteriori. The verification of the new system can be done without looking into the structure of the previously verified program. This makes the method adequate for top-down development of parallel systems. In constrast, the Owicki-Gries method only works for *closed systems*, where all component programs must be simultaneously known. If a new component program is added to the parallel composition, the verification process must be restarted due to the non-compositionality of the interference freedom test.

The chapter is organized as follows: section 4.1 presents the abstract syntax and section 4.2 the operational semantics of the language. Section 4.3 defines the notion of validity of a specification in the rely-guarantee formalism and section 4.4 presents the rules of the proof system. The soundness of the proof system with relation to the underlying semantics is proven in section 4.5. Section 4.6 defines a user-friendly concrete syntax for using the method and section 4.7 shows the applicability of the formalization by verifying several examples. Section 4.8 summarizes the main results.

4.1 Abstract Syntax

Like in the previous formalization, the languages of sequential and parallel programs are defined in different layers.

types
  $\alpha\ bexp = \alpha\ set$
datatype $\alpha\ com =$
    $Basic\ (\alpha \Rightarrow \alpha)$
  $|\ Seq\ (\alpha\ com)\ \ (\alpha\ com)$
  $|\ Cond\ (\alpha\ bexp)\ \ (\alpha\ com)\ \ (\alpha\ com)$
  $|\ While\ (\alpha\ bexp)\ \ (\alpha\ com)$

  | *Await* $(\alpha$ *bexp*$)$ $(\alpha$ *com*$)$
types α *par-com* = α *com option list*

The language of component programs α *com* is a standard sequential while-language augmented with the known synchronization construct *Await*. The only difference with the language of component programs of section 2.1 is the lack of assertions in the syntax. Component programs are not presented as proof outlines, so intermediate assertions can be omitted.

Parallelism is defined in a separate layer with type α *par-com*; it is simply a list of optional component programs. This is analogous to the constructor *Parallel* defined in the Owicki-Gries language. For simplicity, we consider parallelism only at the top level, i.e. no nested parallelism. Moreover, contrary to the formalization in 2.1 parallelism appears as a single construction, i.e. there are neither sequential nor if- nor while-constructions for parallel programs. This way, we reduce the number of constructors and avoid having to duplicate proofs.

Like in chapter 2, this model of parallelism allows composition of any number of sequential component programs by grouping them in a *list*. Representation of parameterized parallel programs may be achieved by means of the function *map*.

4.2 Operational Semantics

The execution of a component program is characterized by an operational semantics that distinguishes between two kinds of transitions: *program* (or *component*) *transitions*, performed by the component itself, and *environment transitions*, performed by another component or an arbitrary environment. The latter affects the state but leaves the program unchanged.

The set of rules defining program transitions is analogous to the rules presented in §2.2. Execution of programs is described via *computations*, which record the sequence of transitions of both kinds. This semantics allows us to define the computation of parallel programs in terms of the computations of the components via a special operator described in section 4.5.2.

Next, we introduce the rules of the semantics and the definition of computation for each layer of the language.

4.2.1 Semantics of Component Programs

A configuration is a pair (P, σ), where P is some program or *None* standing for a terminated program, and σ is a state.

types $\alpha\ conf = \alpha\ com\ option \times \alpha$

A transition is represented by a labelled arrow connecting the beginning and ending configurations. There are two kinds of transitions:

Environment transitions, labelled with e, represent a step from the environment and can only change the state.

consts $etran :: (\alpha\ conf \times \alpha\ conf)\ set$
syntax $\text{-}etran :: \alpha\ conf \Rightarrow \alpha\ conf \Rightarrow bool \qquad (\text{-} -e\rightarrow \text{-})$
translations $P\ -e\rightarrow\ Q \rightleftharpoons (P,\ Q) \in etran$
inductive *etran*
intros
  Env: $(P,\ s)\ -e\rightarrow\ (P,\ t)$

Component transitions, labelled with c, represent a step of a sequential component program

consts $ctran :: (\alpha\ conf \times \alpha\ conf)\ set$
syntax
  $\text{-}ctran :: \alpha\ conf \Rightarrow \alpha\ conf \Rightarrow bool \qquad (\text{-} -c\rightarrow \text{-})$
  $\text{-}ctran^* :: \alpha\ conf \Rightarrow \alpha\ conf \Rightarrow bool \qquad (\text{-} -c{*}\rightarrow \text{-})$
translations
  $P\ -c\rightarrow\ Q \rightleftharpoons (P,\ Q) \in ctran$
  $P\ -c{*}\rightarrow\ Q \rightleftharpoons (P,\ Q) \in ctran^*$

where $P\ -c{*}\rightarrow\ Q$ is the reflexive transitive closure of $P\ -c\rightarrow\ Q$.

inductive *ctran*
intros
  Basic: $(Some\ (Basic\ f),\ s)\ -c\rightarrow\ (None,\ f\ s)$

  *Seq*1: $(Some\ P_0,\ s)\ -c\rightarrow\ (None,\ t)$
    $\Longrightarrow (Some\ (Seq\ P_0\ P_1),\ s)\ -c\rightarrow\ (Some\ P_1,\ t)$
  *Seq*2: $(Some\ P_0,\ s)\ -c\rightarrow\ (Some\ P_2,\ t)$
    $\Longrightarrow (Some\ (Seq\ P_0\ P_1),\ s)\ -c\rightarrow\ (Some\ (Seq\ P_2\ P_1),\ t)$

CondT: $s \in b \Longrightarrow (Some\ (Cond\ b\ P_1\ P_2),\ s) \ -c\rightarrow\ (Some\ P_1,\ s)$
CondF: $s \notin b \Longrightarrow (Some\ (Cond\ b\ P_1\ P_2),\ s) \ -c\rightarrow\ (Some\ P_2,\ s)$

WhileF: $s \notin b \Longrightarrow (Some\ (While\ b\ P),\ s) \ -c\rightarrow\ (None,\ s)$
WhileT: $s \in b \Longrightarrow (Some\ (While\ b\ P),\ s) \ -c\rightarrow\ (Some\ (Seq\ P\ (While\ b\ P)),\ s)$

Await: $[\![\ s \in b;\ (Some\ P,\ s)\ -c{*}\rightarrow\ (None,\ t)\ ]\!]$
$\Longrightarrow (Some\ (Await\ b\ P),\ s)\ -c\rightarrow\ (None,\ t)$

Basic actions and evaluation of boolean conditions are atomic. In both conditional and iteration statements, the evaluation of the boolean tests are atomic, but a step of the environment can interrupt between the boolean test and the first action from the corresponding program body. The body of an await-statement is executed atomically, thus no environment transitions can occur.

It is usual to ensure that await-statements always terminate by disallowing iteration and await-statements in the body, however, this restriction is not necessary for the soundness proofs of this formalization and is thus not required.

4.2.2 Semantics of Parallel Programs

The semantics of parallel programs is also defined by transition rules between configurations. A configuration for a parallel program is a pair formed by a program (α *par-com*) and a state. A parallel program has terminated if so have all its components, i.e. when all component programs are *None*, but this is also of type α *par-com*. Thus, we do not need to wrap the program part into an *option* type.

types
α *par-conf* $= \alpha$ *par-com* $\times\ \alpha$

Transitions may be from the environment, labelled with *pe*, or from the parallel program, labelled with *pc*.

consts
par-etran :: (α *par-conf* $\times\ \alpha$ *par-conf*) *set*
par-ctran :: (α *par-conf* $\times\ \alpha$ *par-conf*) *set*

syntax
  -par-etran:: α *par-conf* $\Rightarrow$ α *par-conf* $\Rightarrow$ *bool*   (- $-pe\rightarrow$ -)
  -par-ctran:: α *par-conf* $\Rightarrow$ α *par-conf* $\Rightarrow$ *bool*   (- $-pc\rightarrow$ -)
translations
  $P\ -pe\rightarrow\ Q\ \rightleftharpoons (P,\ Q) \in$ *par-etran*
  $P\ -pc\rightarrow\ Q\ \rightleftharpoons (P,\ Q) \in$ *par-ctran*

The transition rule for environment transitions is as expected.

inductive *par-etran*
intros
  ParEnv: $(Ps,\ s)\ -pe\rightarrow\ (Ps,\ t)$

The execution of a parallel program is modeled by a nondeterministic interleaving of the atomic actions of the components. In other words, a parallel program performs a component step when one of its non-terminated components performs a component step.

inductive *par-ctran*
intros
  ParComp: $[\![\ i{<}length\ Ps;\ (Ps!i,\ s)\ -c\rightarrow\ (r,\ t)\ ]\!]$
    $\Longrightarrow (Ps,\ s)\ -pc\rightarrow\ (Ps[i{:=}r],\ t)$

$Ps[i{:=}r]$ is the list of programs Ps with the program i replaced by r. This is the only transition rule. If we extend the syntax with other constructors at the parallel level, the set of rules defining the semantics should be augmented with the corresponding rules.

4.2.3 Computations

A computation is defined in [Xu *et al.*, 1997] as any sequence of the form

$$(P_0, \sigma_0) \xrightarrow{\delta_1} (P_1, \sigma_1) \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_n} (P_n, \sigma_n) \xrightarrow{\delta_{n+1}} \ldots,\ \ \delta_i \in \{e, c\}$$

There are many ways of formalizing this concept. Given a definition of computation, the main requirement is to be able to access the program fragment and the state of each configuration, and also the kind of transition between two configurations.

The solution we adopted is to model computations as an inductive set of lists of configurations. The one-element list is always a computation, and two inference rules, one for each kind of transition, determine which lists belong to the inductive set.

types α *confs* = α *conf list*
consts *cptn* :: α *confs set*
inductive *cptn*
intros
  CptnOne: $[(P,\ s)] \in cptn$
  CptnEnv: $(P,\ t)\#xs \in cptn \Longrightarrow (P,\ s)\#(P,\ t)\#xs \in cptn$
  CptnComp: $[\![\ (P,\ s)\ -c\rightarrow\ (Q,\ t);\ (Q,\ t)\#xs \in cptn\ ]\!]$
    $\Longrightarrow (P,\ s)\#(Q,\ t)\#xs \in cptn$

Given two consecutive configurations in a computation it is always possible to determine the kind of transition between them by comparing both program fragments: environment transitions leave the program unchanged while component transitions always change it. Computations of parallel programs are defined analogously.

types α *par-confs* = α *par-conf list*
consts *par-cptn* :: α *par-confs set*
inductive *par-cptn*
intros
  ParCptnOne: $[(P,\ s)] \in par\text{-}cptn$
  ParCptnEnv: $(P,\ t)\#xs \in par\text{-}cptn \Longrightarrow (P,\ s)\#(P,\ t)\#xs \in par\text{-}cptn$
  ParCptnComp: $[\![\ (P,\ s)\ -pc\rightarrow\ (Q,\ t);\ (Q,\ t)\#xs \in par\text{-}cptn\ ]\!]$
    $\Longrightarrow (P,\ s)\#(Q,\ t)\#xs \in par\text{-}cptn$

The set of computations of a program P starting from some initial state s is defined as the set of lists of configurations with first element the pair $(P,\ s)$ which are a computation.

constdefs
  $cp :: \alpha\ com\ option \Rightarrow \alpha \Rightarrow \alpha\ confs\ set$
  $cp\ P\ s \equiv \{l.\ l!0 = (P,\ s) \wedge l \in cptn\}$

  $par\text{-}cp :: \alpha\ par\text{-}com \Rightarrow \alpha \Rightarrow \alpha\ par\text{-}confs\ set$
  $par\text{-}cp\ P\ s \equiv \{l.\ l!0 = (P,\ s) \wedge l \in par\text{-}cptn\}$

4.2.4 Modular Definition of Computation

The definition of computation of sequential programs presented in the previous section follows the one proposed in [Xu *et al.*, 1997] and is probably the most natural and intuitive approach. However, it represents the execution of a program in a simplified linear way without taking into account the inherent structure of the development of a computation.

In the definition of the programming language, however, we observe a well defined structure. For example, the sequential composition is formed from two programs, and the body of a while or an await constructor is itself a program. This structure is automatically reflected in the corresponding computations.

For the proof of some properties, this modular structure is very important. Trying to retrieve this information out of the linear representation of the computation results in tedious and illegible proofs. Such proofs are not appropriate for being carried out in a theorem prover and can often be avoided by redefining concepts. The alternative definition for computations proposed in this section explicitly shows the structure of the program, thus considerably simplifying some proofs, especially those concerning properties of while-programs.

First, we define the auxiliary function *lift* that returns, given a configuration and a program Q, the same configuration where the program has been sequentially composed with Q. If the concerned program is finished, i.e. *None*, the returned program is just Q.

constdefs
  $\mathit{lift} :: \alpha\ \mathit{com} \Rightarrow \alpha\ \mathit{conf} \Rightarrow \alpha\ \mathit{conf}$
  $\mathit{lift}\ Q \equiv \lambda(P,s).\ (\mathit{if}\ P{=}\mathit{None}\ \mathit{then}\ (\mathit{Some}\ Q,\ s)\ \mathit{else}\ (\mathit{Some}(\mathit{Seq}\ (\mathit{the}\ P)\ Q),\ s))$

The set of computations can be defined respecting the modular structure by the following rules:

consts *cptn-mod* :: α *confs set*
inductive *cptn-mod*
intros
  CptnModOne: $[(P,\ s)] \in \mathit{cptn\text{-}mod}$
  CptnModEnv: $(P,\ t)\#xs \in \mathit{cptn\text{-}mod} \Longrightarrow (P,\ s)\#(P,\ t)\#xs \in \mathit{cptn\text{-}mod}$
  CptnModNone: $[\![\ (\mathit{Some}\ P,\ s)\ -c\rightarrow\ (\mathit{None},\ t);\ (\mathit{None},\ t)\#xs \in \mathit{cptn\text{-}mod}\ ]\!]$
    $\Longrightarrow (\mathit{Some}\ P,\ s)\#(\mathit{None},\ t)\#xs \in \mathit{cptn\text{-}mod}$

  CptnModCondT: $[\![\ (\mathit{Some}\ P_0,\ s)\#ys \in \mathit{cptn\text{-}mod};\ s \in b\ ]\!]$
    $\Longrightarrow (\mathit{Some}\ (\mathit{Cond}\ b\ P_0\ P_1),\ s)\#(\mathit{Some}\ P_0,\ s)\#ys \in \mathit{cptn\text{-}mod}$
  CptnModCondF: $[\![\ (\mathit{Some}\ P_1,\ s)\#ys \in \mathit{cptn\text{-}mod};\ s \notin b\ ]\!]$
    $\Longrightarrow (\mathit{Some}\ (\mathit{Cond}\ b\ P_0\ P_1),\ s)\#(\mathit{Some}\ P_1,\ s)\#ys \in \mathit{cptn\text{-}mod}$

$CptnModSeq1$: $\llbracket$ $(Some\ P_0,\ s)\#xs \in cptn\text{-}mod;\ zs = map\ (lift\ P_1)\ xs$ $\rrbracket$
$\qquad\qquad \Longrightarrow (Some\ (Seq\ P_0\ P_1),\ s)\#zs \in cptn\text{-}mod$

$CptnModSeq2$:
$\llbracket$ $(Some\ P_0,\ s)\#xs \in cptn\text{-}mod;\ fst\ (last\ ((Some\ P_0,\ s)\#xs)) = None;$
$(Some\ P_1,\ snd\ (last\ ((Some\ P_0,\ s)\#xs)))\#ys \in cptn\text{-}mod;$
$zs = (map\ (lift\ P_1)\ xs)@ys$ $\rrbracket \Longrightarrow (Some\ (Seq\ P_0\ P_1),\ s)\#zs \in cptn\text{-}mod$

$CptnModWhile1$:
$\llbracket$ $(Some\ P,\ s)\#xs \in cptn\text{-}mod;\ s \in b;\ \ zs = map\ (lift\ (While\ b\ P))\ xs$ $\rrbracket$
$\Longrightarrow (Some\ (While\ b\ P),\ s)\#(Some\ (Seq\ P\ (While\ b\ P)),\ s)\#zs \in cptn\text{-}mod$

$CptnModWhile2$:
$\llbracket$ $(Some\ P,\ s)\#xs \in cptn\text{-}mod;\ \ fst\ (last\ ((Some\ P,\ s)\#xs)) = None;$
$s \in b;\ zs = (map\ (lift\ (While\ b\ P))\ xs)@ys;$
$(Some\ (While\ b\ P),\ snd\ (last\ ((Some\ P,\ s)\#xs)))\#ys \in cptn\text{-}mod$ $\rrbracket$
$\Longrightarrow (Some\ (While\ b\ P),\ s)\#(Some\ (Seq\ P\ (While\ b\ P)),\ s)\#zs \in cptn\text{-}mod$

The first two rules are the same as in the set or rules defining *cptn*. The third rule of *cptn*, namely *CptnComp*, is now replaced by seven rules which not only take into account that the first step is performed by the component program but also consider the kind of program performing the step.

The rule *CptnModNone* summarizes the three possible steps where the program terminates: *Basic*, *WhileF* and *Await*. The two rules for the conditional are obvious. Observe that for these five cases the new definition does not provide any richer information than the *CptnComp* rule with case analysis on the corresponding *c*-step.

The rule *CptnModSeq*1 represents the computations of a sequential composition where execution does not enter the second program, and *Cptn-ModSeq*2 those who at least finish the first program. For while-programs a computation might enter the body but not finish it (*CptnModWhile*1) or finish it at least once (*CptnModWhile*2).

The new definition is useful for proofs about computations of while-programs because, in general, we do not know how often the body is executed. By using rule induction on *cptn-mod* we directly obtain the three following cases:

1. *CptnModNone*: the while-body is not entered.
2. *CptnModWhile*1: the execution of the body is at least started.
3. *CptnModWhile*2: the body is executed completely at least once followed by a new computation of the same while-program, on which the

induction hypothesis holds.

The proof power of applying rule induction to *cptn-mod* is, at least for the while-case, decisive for the proof of soundness in §4.5. In contrast, the information obtained by using the same proof method on *cptn* was almost useless.

Equivalence of both Definitions

The new definition of computation does not represent the intuitive idea of a computation as obviously as the previous one does. The reader might not be convinced that the set generated by these rules contains all computations defined by the set *cptn* and vice versa. For this reason, and also because we still want to use the previous definition when it is convenient, we prove their equivalence in the following theorem:

theorem *cptn-iff-cptn-mod*: $(c \in \mathit{cptn}) = (c \in \mathit{cptn\text{-}mod})$

Proof. The if-direction is fairly easy.

lemma *cptn-if-cptn-mod*: $c \in \mathit{cptn\text{-}mod} \Longrightarrow c \in \mathit{cptn}$

It is proven by rule induction on *cptn-mod.* The only-if-direction is more complicated since it requires recovering the missing structure.

lemma *cptn-onlyif-cptn-mod*: $c \in \mathit{cptn} \Longrightarrow c \in \mathit{cptn\text{-}mod}$

It is proved by rule induction on *cptn*, with a nested structural induction on the program for the case where the first step is made by the component, i.e. we need to prove the following auxiliary lemma by structural induction on a:

lemma *cptn-onlyif-cptn-mod-aux*:
$[\![\ (\mathit{Some}\ a,\ s)\ -c\rightarrow\ (Q,\ t);\ (Q,\ t)\ \#\ xs \in \mathit{cptn\text{-}mod}\ ]\!]$
$\Longrightarrow (\mathit{Some}\ a,\ s)\ \#\ (Q,\ t)\ \#\ xs \in \mathit{cptn\text{-}mod}$

In the proof of this lemma we need an important property stating that the computation of a sequential composition of programs can be divided into a computation of the first program and a computation of the second one.

lemma *div-seq*:
$(\mathit{Some}\ (\mathit{Seq}\ P\ Q),\ s)\ \#\ zs \in \mathit{cptn\text{-}mod} \Longrightarrow$

$$\begin{array}{l}
\exists\, xs.\ (Some\ P,\ s)\ \#\ xs \in cptn\text{-}mod\ \wedge \\
(zs = map\ (lift\ Q)\ xs \vee fst\ (last\ ((Some\ P,\ s)\ \#\ xs)) = None\ \wedge \\
(\exists\, ys.\ (Some\ Q,\ snd\ (last\ ((Some\ P,\ s)\ \#\ xs)))\ \#\ ys \in cptn\text{-}mod \\
\qquad \wedge\ zs = map\ (lift\ Q)\ xs\ @\ ys))
\end{array}$$

The proof is by rule induction on *cptn-mod*. □

4.3 Validity of Correctness Formulas

A rely-guarantee correctness formula (or specification) of a program P consists of the quadruple $(pre,\ rely,\ guar,\ post)$. These four conditions can be classified in two parts:

- *Assumptions*, represented by the pre- and rely condition, describe the conditions under which the program runs, and
- *Commitments*, composed by the guarantee and postcondition, describe the expected behaviors of the program when it is run under the assumptions.

The pre- and postcondition are, like in the traditional Hoare logic, sets of states. They impose conditions upon the initial and final states of a computation, respectively. The rely and guarantee conditions describe properties of environment transitions and transitions of the program, respectively. Thus, they describe sets of pairs of states, formed by the state before and after the transition.

Jones first suggested in [Jones, 1981] that the rely and guarantee conditions be reflexive and transitive. However, for the soundness proof only the reflexivity of the guarantee condition is necessary.

4.3.1 Validity for Component Programs

Specifications of sequential programs are written with the syntax "P *sat* $[pre,\ rely,\ guar,\ post]$" where *sat* stands for "satisfies". The type of these tuples is:

types $\alpha\ rgformula = \alpha\ com \times \alpha\ set \times (\alpha \times \alpha)\ set \times (\alpha \times \alpha)\ set \times \alpha\ set$

Informally, we say that P satisfies its specification if under the assumptions that

1 P is started in a state that satisfies *pre*, and

2 any environment transition in the computation satisfies *rely*,

then P ensures the following commitments:

3 any component transition satisfies *guar*, and

4 if the computation terminates, the final state satisfies *post*.

The formal definitions are given by the functions:

constdefs
  $assum :: (\alpha\ set \times (\alpha \times \alpha)\ set) \Rightarrow \alpha\ confs\ set$
  $assum \equiv \lambda(pre,\ rely)\ .\ \{c.\ snd\ (c!0) \in pre \wedge (\forall i.\ Suc\ i{<}length\ c \longrightarrow$
  $\qquad c!i\ {-e\rightarrow}\ c!Suc\ i \longrightarrow (snd\ (c!i),\ snd\ (c!Suc\ i)) \in rely)\}$

  $comm :: ((\alpha \times \alpha)\ set \times \alpha\ set) \Rightarrow \alpha\ confs\ set$
  $comm \equiv \lambda(guar,\ post)\ .\ \{c.\ (\forall i.\ Suc\ i{<}length\ c \longrightarrow$
  $\qquad c!i\ {-c\rightarrow}\ c!Suc\ i \longrightarrow (snd\ (c!i),\ snd\ (c!Suc\ i)) \in guar) \wedge$
  $\qquad (fst\ (last\ c) = None \longrightarrow snd\ (last\ c) \in post)\}$

A rely-guarantee specification of a sequential component program P is valid, and we use the usual syntax $\models P\ sat\ [pre,\ rely,\ guar,\ post]$, iff for any initial state, all computations of P that satisfy the assumptions satisfy the commitments.

constdefs
  $com\text{-}validity :: \alpha\ com \Rightarrow \alpha\ set \Rightarrow (\alpha \times \alpha)\ set \Rightarrow (\alpha \times \alpha)\ set \Rightarrow \alpha\ set \Rightarrow bool$
  $\qquad (\models \text{-}\ sat\ [\text{-},\ \text{-},\ \text{-},\ \text{-}]\ )$
  $\models P\ sat\ [pre,\ rely,\ guar,\ post] \equiv$
  $\forall s.\ cp\ (Some\ P)\ s \cap assum\ (pre,\ rely) \subseteq comm\ (guar,\ post)$

4.3.2 Validity for Parallel Programs

Parallel programs can be seen as a unit executed in a possibly interfering environment. For this reason, we include a rely and a guarantee condition in the specification of parallel programs as well. They have the form $P\ SAT\ [pre,\ rely,\ guar,\ post]$ where P has the type $\alpha\ par\text{-}com$.

A parallel program has finished when all its components are *None*. To abbreviate this we introduce the following definition:

constdefs
  $All\text{-}None :: \alpha\ com\ option\ list \Rightarrow bool$
  $All\text{-}None\ xs \equiv \forall c{\in}set\ xs.\ c = None$

The definition of assumptions, commitments and validity are analogous to the previous section:

constdefs
$\mathit{par\text{-}assum} :: (\alpha\ \mathit{set} \times (\alpha \times \alpha)\ \mathit{set}) \Rightarrow \alpha\ \mathit{par\text{-}confs}\ \mathit{set}$
$\mathit{par\text{-}assum} \equiv \lambda(\mathit{pre}, \mathit{rely}).\ \{c.\ \mathit{snd}\ (c!0) \in \mathit{pre} \wedge (\forall i.\ \mathit{Suc}\ i{<}\mathit{length}\ c \longrightarrow$
$\quad c!i\ {-pe\rightarrow}\ c!\mathit{Suc}\ i \longrightarrow (\mathit{snd}\ (c!i),\ \mathit{snd}\ (c!\mathit{Suc}\ i)) \in \mathit{rely})\}$

$\mathit{par\text{-}comm} :: ((\alpha \times \alpha)\ \mathit{set} \times \alpha\ \mathit{set}) \Rightarrow \alpha\ \mathit{par\text{-}confs}\ \mathit{set}$
$\mathit{par\text{-}comm} \equiv \lambda(\mathit{guar}, \mathit{post}).\ \{c.\ (\forall i.\ \mathit{Suc}\ i{<}\mathit{length}\ c \longrightarrow$
$\quad c!i\ {-pc\rightarrow}\ c!\mathit{Suc}\ i \longrightarrow (\mathit{snd}\ (c!i),\ \mathit{snd}\ (c!\mathit{Suc}\ i)) \in \mathit{guar}) \wedge$
$\quad (\mathit{All\text{-}None}\ (\mathit{fst}\ (\mathit{last}\ c)) \longrightarrow \mathit{snd}\ (\mathit{last}\ c) \in \mathit{post})\}$

$\mathit{par\text{-}com\text{-}validity} :: \alpha\ \ \mathit{par\text{-}com} \Rightarrow \alpha\ \mathit{set} \Rightarrow (\alpha \times \alpha)\ \mathit{set} \Rightarrow (\alpha \times \alpha)\ \mathit{set} \Rightarrow \alpha\ \mathit{set}$
$\quad \Rightarrow \mathit{bool}\qquad (\models\ \text{-}\ \mathit{SAT}\ [\text{-}, \text{-}, \text{-}, \text{-}]\ )$
$\models P\ \mathit{SAT}\ [\mathit{pre}, \mathit{rely}, \mathit{guar}, \mathit{post}] \equiv$
$\quad \forall s.\ \mathit{par\text{-}cp}\ P\ s \cap \mathit{par\text{-}assum}\ (\mathit{pre}, \mathit{rely}) \subseteq \mathit{par\text{-}comm}\ (\mathit{guar}, \mathit{post})$

4.4 The Proof System

The system of axioms and inference rules for deriving partial correctness formulas of parallel programs in the rely-guarantee formalism can be regarded as a compositional reformulation of the Owicki-Gries system. Due to the layered definition of the syntax, the set of all derivable specifications is defined using two sets:

1. The set of all derivable specifications of sequential programs: *rghoare*.
2. The set of all derivable specifications of parallel programs: *par-rghoare*.

The definition of the second set uses the previous one. Thus, the declarations must follow the previous order.

4.4.1 Proof System for Component Programs

We first define a predicate about stability needed in the rules.

constdefs
$\mathit{stable} :: \alpha\ \mathit{set} \Rightarrow (\alpha \times \alpha)\ \mathit{set} \Rightarrow \mathit{bool}$
$\mathit{stable} \equiv \lambda f\ g.\ \forall x\ y.\ x \in f \longrightarrow (x,\ y) \in g \longrightarrow y \in f$

For example, *stable pre rely* means that if a state belongs to the precondition and some transition satisfies the rely condition, then the reached state still belongs to the precondition.

The set of all derivable specifications is defined by the constant

consts *rghoare* :: α *rgformula set*

where α *rgformula* is the type of a specification of a sequential component program (see §4.3.1). A derivable specification is denoted with the usual syntax:

syntax
  -rghoare :: α *com* $\Rightarrow$ α *set* $\Rightarrow$ $(\alpha \times \alpha)$ *set* $\Rightarrow$ $(\alpha \times \alpha)$ *set* $\Rightarrow$ α *set* $\Rightarrow$ *bool*
    ($\vdash$ - *sat* [-, -, -, -])
translations
  $\vdash$ *P sat* [*pre*, *rely*, *guar*, *post*] $\rightleftharpoons$ (*P*, *pre*, *rely*, *guar*, *post*) $\in$ *rghoare*

We follow [Xu *et al.*, 1997] in the definition of the rules, but differ mainly in the representation of the variables. In [Xu *et al.*, 1997], rules are expressed in terms of a variable y representing the vector of program variables and the corresponding primed variable y' referring to the same vector after a transformation. In our formalization we describe properties of states or of pairs of states by directly describing the set of tuples of values, i.e. we do not refer to program variables. For example, the set of pairs of states representing the identity transformation is $\{(s,t).\ s = t\}$, whereas in [Xu *et al.*, 1997] it would be $y = y'$.

inductive *rghoare*
intros
  Basic: $[\![$ *pre* $\subseteq \{s.\ f\ s \in$ *post*$\}$; $\{(s,\ t).\ s \in$ *pre* $\wedge\ (t = f\ s \vee t = s)\} \subseteq$ *guar*;
    stable pre rely; *stable post rely* $]\!]$
    $\Longrightarrow \vdash$ *Basic f sat* [*pre*, *rely*, *guar*, *post*]

In the computation of a *Basic* command there is exactly one component transition that updates the state. Before and after this component transition there can be a number of environment transitions. The initial state satisfies *pre*, thus from *stable pre rely* it follows that *pre* holds immediately before the component transition takes place. From *pre* $\subseteq \{s.\ f\ s \in$ *post*$\}$ it follows that *post* holds immediately after the component transition, and because *post* is stable when *rely* holds, *post* holds after any number of environment transitions.
The rules for the sequential composition and conditional statements are standard:

Seq: $[\![\vdash P\ sat\ [pre,\ rely,\ guar,\ mid];\ \vdash Q\ sat\ [mid,\ rely,\ guar,\ post]\ ]\!]$
$\implies \vdash Seq\ P\ Q\ sat\ [pre,\ rely,\ guar,\ post]$

Cond: $[\![\ stable\ pre\ rely;\ \vdash P_1\ sat\ [pre \cap b,\ rely,\ guar,\ post];$
$\vdash P_2\ sat\ [pre \cap -b,\ rely,\ guar,\ post];\ \forall s.\ (s,\ s){\in}guar\ ]\!]$
$\implies \vdash Cond\ b\ P_1\ P_2\ sat\ [pre,\ rely,\ guar,\ post]$

In the while-rule the precondition plays the role of the invariant; it must hold before and after execution of the body at every iteration:

While: $[\![\ stable\ pre\ rely;\ pre \cap -b \subseteq post;\ stable\ post\ rely;$
$\vdash P\ sat\ [pre \cap b,\ rely,\ guar,\ pre];\ \forall s.\ (s,\ s){\in}guar\ ]\!]$
$\implies \vdash While\ b\ P\ sat\ [pre,\ rely,\ guar,\ post]$

The rule for the await-statement is less obvious:

Await: $[\![\ \forall V.\ \vdash P\ sat\ [pre \cap b \cap \{V\},\ \{(s,\ t).\ s = t\},\ UNIV,$
$\{s.\ (V,\ s) \in guar\} \cap post];\ stable\ pre\ rely;\ stable\ post\ rely\ ]\!]$
$\implies \vdash Await\ b\ P\ sat\ [pre,\ rely,\ guar,\ post]$

By the semantics of the await-command, a positive evaluation of the condition and the execution of the body is done atomically. Thus, the state transition caused by the complete execution of P must satisfy the guarantee condition. This is reflected in the precondition and postcondition of P in the assumptions; since these are sets of single states, the relation between the state before and after the transformation is established by fixing the values of the first via a universally quantified variable V. The intermediate state changes during the execution of P must not guarantee anything, thus the guarantee condition is the universal set *UNIV*. However, since they are executed atomically, the environment cannot change their values. This is reflected by the rely condition $\{(s,\ t).\ s = t\}$. To ensure that the postcondition holds at the end of the computation, regardless of possible environment transitions, we require *stable post rely*.
Finally, the rule of consequence allows to strengthen the assumptions and weaken the commitments:

Conseq: $[\![\ pre \subseteq pre';\ rely \subseteq rely';\ guar' \subseteq guar;\ post' \subseteq post;$
$\vdash P\ sat\ [pre',\ rely',\ guar',\ post']\ ]\!]$
$\implies \vdash P\ sat\ [pre,\ rely,\ guar,\ post]$

These six rules inductively define the set of derivable specifications of sequential component programs. In §4.5 we prove that only valid specifications can be derived, i.e. we prove the soundness of the system.

The functions defined below extract the parts of a specification of a sequential parallel program and will be used in the following section:

constdefs
$Pre :: \alpha\ rgformula \Rightarrow \alpha\ set$
$Pre\ x \equiv fst\ (snd\ x)$
$Post :: \alpha\ rgformula \Rightarrow \alpha\ set$
$Post\ x \equiv snd\ (snd\ (snd\ (snd\ x)))$
$Rely :: \alpha\ rgformula \Rightarrow (\alpha \times \alpha)\ set$
$Rely\ x \equiv fst\ (snd\ (snd\ x))$
$Guar :: \alpha\ rgformula \Rightarrow (\alpha \times \alpha)\ set$
$Guar\ x \equiv fst\ (snd\ (snd\ (snd\ x)))$
$Com :: \alpha\ rgformula \Rightarrow \alpha\ com$
$Com\ x \equiv fst\ x$

4.4.2 Proof System for Parallel Programs

This section presents the rule for deriving parallel programs whose components are sequential. Observe that in the definition of validity for parallel programs (see §4.3.2) no information about the pre, post, rely and guarantee conditions of the component programs was included. This was not important for the definition of validity, however, at the level of concrete verification of programs with the system of rules, we want to apply the rules backwards. Therefore, the conclusion should include all the information needed in the premises of the rule. For this reason, we include it as part of the elements of the set of derivable formulas. Their type is

types $\alpha\ par\text{-}rgformula = \alpha\ rgformula\ list \times \alpha\ set \times (\alpha \times \alpha)\ set \times (\alpha \times \alpha)\ set \times \alpha\ set$

The type *α rgformula* corresponds to a full specification of a component program (see §4.3.1). The constant defining the corresponding set of derivations, called *par-rghoare*, and a familiar syntax for membership of an element are shown below.

consts $par\text{-}rghoare :: \alpha\ par\text{-}rgformula\ set$
syntax
$\text{-}par\text{-}rghoare :: \alpha\ rgformula\ list \Rightarrow \alpha\ set \Rightarrow (\alpha \times \alpha)\ set \Rightarrow (\alpha \times \alpha)\ set \Rightarrow \alpha\ set \Rightarrow bool\ \ (\vdash\ \text{-}\ SAT\ [\text{-}, \text{-}, \text{-}, \text{-}]\ )$

translations
$\vdash Ps\ SAT\ [pre,\ rely,\ guar,\ post] \rightleftharpoons (Ps,\ pre,\ rely,\ guar,\ post) \in par\text{-}rghoare$

Do not confuse the type of the first argument. Here it is a list of specifications of sequential component programs, while in $\models P\ SAT\ [pre,\ rely,\ guar,\ post]$, P is just a program of type α *par-com*.

The rule for parallel composition is new in the sense that it generalizes the case of composing two programs, as given in [Xu *et al.*, 1997, de Roever *et al.*, 2000], to the generic case of composing any number of programs. This rule allows the verification of parameterized parallel programs directly in the system.

inductive *par-rghoare*
intros

$$
\begin{array}{l}
Parallel: \\
[\![\ \forall\, i{<}length\ Ps.\ rely \cup (\bigcup j{\in}\{j.\ j{<}length\ Ps \wedge j{\neq}i\}.\ Guar\ (Ps!j)) \subseteq Rely\ (Ps!i); \\
\quad (\bigcup j{\in}\{j.\ j{<}length\ Ps\}.\ Guar\ (Ps!j)) \subseteq guar; \\
\quad pre \subseteq (\bigcap i{\in}\{i.\ i{<}length\ Ps\}.\ Pre\ (Ps!i)); \\
\quad (\bigcap i{\in}\{i.\ i{<}length\ Ps\}.\ Post\ (Ps!i)) \subseteq post; \\
\forall\, i{<}length\ Ps.\ \vdash Com(Ps!i)\ sat\ [Pre(Ps!i),\ Rely(Ps!i),\ Guar(Ps!i),\ Post(Ps!i)]\,]\!] \\
\quad \Longrightarrow\ \vdash Ps\ SAT\ [pre,\ rely,\ guar,\ post]
\end{array}
$$

An environment transition for the component specified by $Ps!i$ consists of a component transition from any of the other processes $Ps!j$ where $i{\neq}j$, or of a transition from the overall environment. Hence, the strongest rely condition that component i can assume is $rely \cup (\bigcup j{\in}\{j.\ j{<}length\ Ps \wedge j{\neq}i\}.\ Guar\ (Ps!j))$.

A component transition of the parallel program is performed by one of its components, hence they all have to satisfy the overall guarantee condition *guar*. Like in the Owicki-Gries system the precondition for the parallel composition must imply all the component's preconditions and the overall postcondition must be a logical consequence of all postconditions.

Finally, the specifications of the components have to be derivable in their system. Consequently, the soundness of the system of rules for sequential component programs is a necessary previous result for the proof of soundness of this rule.

4.5 Soundness

In this section we prove soundness of the rule for parallel composition. For this result the proof of soundness of the system for sequential programs is required.

As explained in §2.5, soundness of a system of rules that inductively define a set of correctness formulas can be shown by rule induction. This proof principle works by proving that a certain property of the elements of the set is true of all axioms and is preserved by each inference rule. Then by construction of the set, we obtain the property for any element in the set. When the property concerned is validity of a correctness formula, then any derivable formula is correct and we say that the system of rules is sound.

Using rule induction soundness of the system amounts to proving soundness of each rule. Thus, for each rule we assume that the formulas that appear in the premises are valid and we must prove that the formula in the conclusion is also valid.

4.5.1 Soundness of the System for Component Programs

We want to prove the theorem:

theorem *rgsound*:
$\vdash P\ sat\ [pre,\ rely,\ guar,\ post] \Longrightarrow \models P\ sat\ [pre,\ rely,\ guar,\ post]$

The proof proceeds by rule induction. This results in six subgoals, one for each rule.

Soundness of the Basic Rule

The rule for a *Basic* command is an axiom. Hence, validity must follow directly from the premises of the rule without any induction hypothesis.

lemma *Basic-sound*:
$[\![\ pre \subseteq \{s.\ f\ s \in post\};\ \{(s,\ t).\ s \in pre \wedge (t = f\ s \vee t = s)\} \subseteq guar;$
$stable\ pre\ rely;\ stable\ post\ rely\ ]\!] \Longrightarrow\ \models Basic\ f\ sat\ [pre,\ rely,\ guar,\ post]$

For this and some of the following proofs we need a lemma about the stability predicate.

Assume a computation whose environment transitions satisfy the rely condition and where all transitions of the subcomputation between two indices j and k are made by the environment. If the state of the configuration at index j satisfies some condition p such that *stable p rely*, then the state at configuration k also satisfies the condition p and the program part is left unchanged:

lemma *stability*:
$\llbracket\ x \in cptn;\ stable\ p\ rely;\ j \leq k;\ k < length\ x;\ snd\ (x!j) \in p;$
$\forall i.\ Suc\ i{<}length\ x \longrightarrow x!i\ {-e\rightarrow}\ x!Suc\ i \longrightarrow (snd\ (x!i),\ snd\ (x\ !\ Suc\ i)){\in}rely;$
$\forall i.\ j \leq i \wedge i < k \longrightarrow x!i\ {-e\rightarrow}\ x\ !\ Suc\ i\ \rrbracket$
$\Longrightarrow snd\ (x!k) \in p \wedge fst\ (x!j) = fst\ (x!k)$

The soundness of the Basic rule is easy to prove with help of *stability* and two more lemmas. The first one states that if there is a component transition in the computation of a *Basic*-command, then it is the only one:

lemma *unique-ctran-Basic*:
$\llbracket\ x \in cptn;\ x!0 = (Some\ (Basic\ f),\ s);\ Suc\ i < length\ x;$
$x!i\ {-c\rightarrow}\ x\ !\ Suc\ i;\ Suc\ j < length\ x;\ i \neq j\ \rrbracket \Longrightarrow x!j\ {-e\rightarrow}\ x\ !\ Suc\ j$

The second one ensures that if the empty program appears in a computation of a Basic-command at some point, then there must be a component transition before:

lemma *exists-ctran-Basic-None*:
$\llbracket\ x \in cptn;\ x!0 = (Some\ (Basic\ f),\ s);\ i < length\ x;\ fst\ (x!i) = None\ \rrbracket$
$\Longrightarrow \exists j.\ j < i \wedge x!j\ {-c\rightarrow}\ x\ !\ Suc\ j$

Both lemmas are proven by induction on the length of the computation.

Soundness of the Await Rule

The induction hypothesis is applied to the derivability of the await-body, thus we can assume that its specification is valid:

lemma *Await-sound*:
$\llbracket\ stable\ pre\ rely;\ stable\ post\ rely;$
$\forall V.\ \models P\ sat\ [pre \cap b \cap \{s.\ s = V\},\ \{(s,\ t).\ s = t\},\ UNIV,$
$\{s.\ (V,\ s){\in}guar\} \cap post]\ \rrbracket$
$\Longrightarrow \models Await\ b\ P\ sat\ [pre,\ rely,\ guar,\ post]$

The proof is similar to the previous one and requires the analogous lemmas:

lemma *unique-ctran-Await*:
$[\![\ x \in cptn;\ x!0 = (Some\ (Await\ b\ c),\ s);\ Suc\ i < length\ x;$
$x!i\ {-c}{\to}\ x\ !\ Suc\ i;\ Suc\ j < length\ x;\ i \neq j\ ]\!] \Longrightarrow x!j\ {-e}{\to}\ x\ !\ Suc\ j$

lemma *exists-ctran-Await-None*:
$[\![\ x \in cptn;\ x!0 = (Some\ (Await\ b\ c),\ s);\ i < length\ x;\ fst\ (x!i) = None\ ]\!]$
$\Longrightarrow \exists j.\ j < i \wedge x!j\ {-c}{\to}\ x\ !\ Suc\ j$

We also need to prove that there is a computation of the body of the await-statement that satisfies the specification given in the premises. However, the only information we obtain from the semantics about the execution of the body is that it terminates in some number of component transitions. We prove that a sequence of component transitions can also be described as a computation:

lemma *Star-imp-cptn*: $(P,\ s)\ {-c{*}}{\to}\ (R,\ t) \Longrightarrow \exists\, l \in cp\ P\ s.\ last\ l = (R,\ t)$

Soundness of the Conditional Rule

Given valid subspecifications of the if- and else-branches we prove that the correctness formula for the conditional statement is also valid:

lemma *Cond-sound*:
$[\![\ stable\ pre\ rely;\ \models P_1\ sat\ [pre \cap b,\ rely,\ guar,\ post];$
$\models P_2\ sat\ [pre \cap -\ b,\ rely,\ guar,\ post];\ \forall s.\ (s,\ s){\in}guar\ ]\!]$
$\Longrightarrow\ \models (Cond\ b\ P_1\ P_2)\ sat\ [pre,\ rely,\ guar,\ post]$

In the proof we first distinguish whether a computation of $(Cond\ b\ P_1\ P_2)$ contains a component transition or not. If not, by the following lemma all transitions are performed by the environment:

lemma *etran-or-ctran*:
$[\![\ x \in cptn;\ m \leq length\ x;\ \forall i.\ Suc\ i < m \longrightarrow \neg\ x!i\ {-c}{\to}\ x\ !\ Suc\ i;\ Suc\ i < m\ ]\!]$
$\Longrightarrow x!i\ {-e}{\to}\ x\ !\ Suc\ i$

Thus the first part of the commitments is trivially fulfilled. By using the *stability* lemma the last program of the computation cannot be *None*, so the the second part of the commitments holds too.

If there is a component transition, then by the lemma

lemma *Ex-first-occurrence*: $P\ n \Longrightarrow \exists m.\ P\ m \wedge (\forall i < m.\ \neg\ P\ i)$

there is a first one. This component transition satisfies the guarantee condition because of the required reflexivity property. By the stability lemma, the precondition holds after this step. Depending on whether the boolean condition of the if-statement holds or not, the fulfillment of the commitments for the rest of the computation follows from the induction hypothesis on the corresponding program.

Soundness of the Sequential Rule

Validity of a specification for sequential composition follows from the validity of appropriate subspecifications of the programs that are sequentially composed:

lemma *Seq-sound*:
$\llbracket \models P\ \mathit{sat}\ [\mathit{pre}, \mathit{rely}, \mathit{guar}, \mathit{mid}];\ \models Q\ \mathit{sat}\ [\mathit{mid}, \mathit{rely}, \mathit{guar}, \mathit{post}]\ \rrbracket$
$\Longrightarrow\ \models \mathit{Seq}\ P\ Q\ \mathit{sat}\ [\mathit{pre}, \mathit{rely}, \mathit{guar}, \mathit{post}]$

In the proof we distinguish whether a computation of the sequential composition finishes computing P or not:

1. If not, we have $\forall\, i{<}\mathit{length}\ x.\ \mathit{fst}\ (x!i){\neq}\mathit{Some}\ Q$. In this case, we can find a computation of P such that c is just the corresponding "lifted" computation.

2. Otherwise, we have $\exists\, i{<}\mathit{length}\ x.\ \mathit{fst}\ (x!i) = \mathit{Some}\ Q$. Such a configuration can occur several times but we are only interested in the first occurrence. The computation c in this case can be split into a "lifted" terminated computation of P followed by a computation of Q.

The following lemmas establish these properties.

lemma *Seq-sound1*:
$\llbracket\ x \in \mathit{cptn\text{-}mod};\ x!0 = (\mathit{Some}\ (\mathit{Seq}\ P\ Q),\ s);\ \forall\, i{<}\mathit{length}\ x.\ \mathit{fst}\ (x!i){\neq}\mathit{Some}\ Q\ \rrbracket$
$\Longrightarrow \exists\, \mathit{xs} \in \mathit{cp}\ (\mathit{Some}\ P)\ s.\ x = \mathit{map}\ (\mathit{lift}\ Q)\ \mathit{xs}$

We prove it by rule induction on *cptn-mod*. The induction hypothesis is only needed at the *CptnModEnv* rule.

lemma *Seq-sound2*:
$\llbracket\ x \in \mathit{cptn};\ x!0 = (\mathit{Some}\ (\mathit{Seq}\ P\ Q),\ s);\ i{<}\mathit{length}\ x;\ \mathit{fst}\ (x!i) = \mathit{Some}\ Q;$
$\forall\, j{<}i.\ \mathit{fst}\ (x!j){\neq}\mathit{Some}\ Q\ \rrbracket$

$\Longrightarrow \exists\, xs\ ys.\ xs \in cp\ (Some\ P)\ s \wedge length\ xs = Suc\ i\ \wedge$
$ys \in cp\ (Some\ Q)\ (snd\ (xs\ !i)) \wedge x = (map\ (lift\ Q)\ xs)@tl\ ys$

This second lemma is easier to prove by rule induction on *cptn*. When the computation results from the rule *CptnComp*, we perform case analysis on the *c*-step. After simplifying, only the cases corresponding to *Seq*1 and *Seq*2 remain. The first one is solved easily without need of the induction hypothesis. The proof for the second one is also straightforward because the induction hypothesis can be directly used.

Back to the main lemma, suppose that a computation of *Seq P Q*, say *c*, satisfies the assumptions *assum* (*pre*, *rely*). If the computation does not finish computing *P*, then it does not terminate, so we only have to prove that all component transitions satisfy *guar*. This follows from $\models P\ sat\ [pre,\ rely,\ guar,\ mid]$ and the lemma *Seq-sound*1. In the second case, we first prove the same thing for the first part of the computation. Then, since it terminates we obtain from $\models P\ sat\ [pre,\ rely,\ guar,\ mid]$ that the last state satisfies *mid*. From $\models Q\ sat\ [mid,\ rely,\ guar,\ post]$ the rest of the computation also satisfies the commitments.

Soundness of the While Rule

The subsequent proof is the most interesting one so far.

lemma *While-sound*:
$[\![\ stable\ pre\ rely;\ pre \cap -\ b \subseteq post;\ stable\ post\ rely;$
$\models P\ sat\ [pre \cap b,\ rely,\ guar,\ pre];\ \forall s.\ (s,\ s) \in guar\ ]\!]$
$\Longrightarrow\ \models While\ b\ P\ sat\ [pre,\ rely,\ guar,\ post]$

First attempts at proving this lemma using the “flat” definition of computation turn out to be unnecessarily cumbersome. Finding a suitable definition of computation together with the proof of equivalence (§4.2.4) and finally the soundness proof of the rule took about a month of work. This seems unnecessary considering that the proof on paper from [Xu *et al.*, 1995] is barely a page and very intuitive. However, some intuitive ideas like the recursive structure of a while-computation are easy to state on paper but turn out to be difficult to formalize in a theorem prover if the definitions are inadequate.

The proof of soundness directly follows from a lemma which can be proved by rule induction on *cptn-mod*:

lemma *While-sound-aux*:
$\llbracket$ *stable pre rely*; *pre* $\cap -$ *b* $\subseteq$ *post*; *stable post rely*;
$\models$ *P sat* [*pre* $\cap$ *b*, *rely*, *guar*, *pre*]; $\forall$ *s*. (*s*, *s*) $\in$ *guar*; *x* $\in$ *cptn-mod* $\rrbracket$
$\Longrightarrow$ $\forall$ *s xs*. *x* = (*Some* (*While b P*), *s*) # *xs* $\longrightarrow$ *x* $\in$ *assum* (*pre*, *rely*)
$\longrightarrow$ *x* $\in$ *comm* (*guar*, *post*)

After some simplification four subgoals remain. We briefly explain their proofs.

The first one corresponds to the *CptnModEnv* rule. After an environment step the program fragment is still a while-program so the induction hypothesis can be applied:

lemma *WhileEnv*:
$\llbracket$ *stable pre rely*; *pre* $\cap -$ *b* $\subseteq$ *post*; *stable post rely*; $\forall$ *s*. (*s*, *s*) $\in$ *guar*;
$\models$ *P sat* [*pre* $\cap$ *b*, *rely*, *guar*, *pre*]; (*Some* (*While b P*), *t*) # *xs* $\in$ *cptn-mod*;
(*Some* (*While b P*), *t*) # *xs* $\in$ *assum* (*pre*, *rely*) $\longrightarrow$
(*Some* (*While b P*), *t*) # *xs* $\in$ *comm* (*guar*, *post*);
(*Some* (*While b P*), *s*) # (*Some* (*While b P*), *t*) # *xs* $\in$ *assum* (*pre*, *rely*) $\rrbracket$
$\Longrightarrow$ (*Some* (*While b P*), *s*) # (*Some* (*While b P*), *t*) # *xs* $\in$ *comm* (*guar*, *post*)

The proof follows directly using the following two lemmas:

lemma *etran-in-comm*: (*P*, *t*) # *xs* $\in$ *comm* (*guar*, *post*)
$\Longrightarrow$ (*P*, *s*) # (*P*, *t*) # *xs* $\in$ *comm* (*guar*, *post*)

lemma *tl-of-assum-in-assum*:
$\llbracket$ (*P*, *s*) # (*P*, *t*) # *xs* $\in$ *assum* (*pre*, *rely*); *stable pre rely* $\rrbracket$
$\Longrightarrow$ (*P*, *t*) # *xs* $\in$ *assum* (*pre*, *rely*)

The second subgoal corresponds to the *CptnModNone* rule, i.e. the while-program terminates:

lemma *WhileNone*:
$\llbracket$ *stable pre rely*; *pre* $\cap -$ *b* $\subseteq$ *post*; *stable post rely*; $\forall$ *s*. (*s*, *s*) $\in$ *guar*;
$\models$ *P sat* [*pre* $\cap$ *b*, *rely*, *guar*, *pre*]; (*Some* (*While b P*), *s*) $-c\rightarrow$ (*None*, *t*);
(*None*, *t*) # *xs* $\in$ *cptn-mod*;
(*Some* (*While b P*), *s*) # (*None*, *t*) # *xs* $\in$ *assum* (*pre*, *rely*) $\rrbracket$
$\Longrightarrow$ (*Some* (*While b P*), *s*) # (*None*, *t*) # *xs* $\in$ *comm* (*guar*, *post*)

By rule inversion on the *c*-transition we obtain $s \notin b$ and $s = t$. The first transition of the computation in the conclusion satisfies the guarantee condition by the reflexivity assumption. The rest of the transitions are environmental. By the assumption $pre \cap - b \subseteq post$ the state s satisfies the postcondition, since *stable post rely* it follows by the *stability* lemma that the final state satisfies the postcondition.

The third subgoal corresponds to the rule *CptnModWhile*1. There are not subcomputations of while-programs in the premises, thus the induction hypothesis cannot be used:

lemma *While*1:
$[\![$ *stable pre rely*; $pre \cap - b \subseteq post$; *stable post rely*; $\forall s.\ (s, s) \in guar$;
$\models$ *P sat* [$pre \cap b$, *rely*, *guar*, *pre*]; (*Some P*, *s*) # *xs* $\in$ *cptn-mod*; $s \in b$;
(*Some* (*While b P*), *s*) # (*Some* (*Seq P* (*While b P*)), *s*) #
map (*lift* (*While b P*)) *xs* $\in$ *assum* (*pre*, *rely*) $]\!]$
$\Longrightarrow$ (*Some* (*While b P*), *s*) # (*Some* (*Seq P* (*While b P*)), *s*) #
map (*lift* (*While b P*)) *xs* $\in$ *comm* (*guar*, *post*)

This kind of computation does not terminate, thus we only have to prove that all component transitions satisfy the guarantee condition. The first component transition is easy due to the reflexivity property. The rest of the computation can be reduced to a computation of P where the initial state satisfies $pre \cap b$, thus the proof follows from $\models$ *P sat* [$pre \cap b$, *rely*, *guar*, *pre*].

The interesting case is the fourth subgoal. The computation contains a full computation of the body followed by a computation of the same while-command, where the induction hypothesis is applied:

lemma *While*2:
$[\![$ *stable pre rely*; $pre \cap - b \subseteq post$; *stable post rely*; $\forall s.\ (s, s) \in guar$;
$\models$ *P sat* [$pre \cap b$, *rely*, *guar*, *pre*]; *fst* (*last* ((*Some P*, *s*) # *xs*)) = *None*;
(*Some P*, *s*) # *xs* $\in$ *cptn-mod*; $s \in b$;
(*Some* (*While b P*), *snd* (*last* ((*Some P*, *s*) # *xs*))) # *ys* $\in$ *cptn-mod*;
(*Some* (*While b P*), *snd* (*last* ((*Some P*, *s*) # *xs*))) # *ys* $\in$ *assum* (*pre*, *rely*)
$\longrightarrow$ (*Some* (*While b P*), *snd* (*last* ((*Some P*, *s*) # *xs*))) # *ys*
$\in$ *comm* (*guar*, *post*);
(*Some* (*While b P*), *s*) # (*Some* (*Seq P* (*While b P*)), *s*) #
map (*lift* (*While b P*)) *xs* @ *ys* $\in$ *assum* (*pre*, *rely*) $]\!]$
$\Longrightarrow$ (*Some* (*While b P*), *s*) # (*Some* (*Seq P* (*While b P*)), *s*) #
map (*lift* (*While b P*)) *xs* @ *ys* $\in$ *comm* (*guar*, *post*)

The first part of the computation, i.e. (*Some* (*While b P*), *s*) # (*Some* (*Seq P* (*While b P*)), *s*) # *map* (*lift* (*While b P*)) *xs* represents the first entire execution of the body. Like for the proof of the last subgoal it follows that the commitments are satisfied by this subcomputation. For the rest of the computation it suffices to apply the induction hypothesis. Thus, we need to prove that it satisfies the assumptions:

lemma *assum-after-body*:
$[\![\models P\ sat\ [pre \cap b,\ rely,\ guar,\ pre];\ (Some\ P,\ s)\ \#\ xs \in cptn\text{-}mod;$
$fst\ (last\ ((Some\ P,\ s)\ \#\ xs)) = None;\ s \in b;$
$(Some\ (While\ b\ P),\ s)\ \#\ (Some\ (Seq\ P\ (While\ b\ P)),\ s)\ \#$
$map\ (lift\ (While\ b\ P))\ xs\ @\ ys \in assum\ (pre,\ rely)\ ]\!]$
$\Longrightarrow (Some\ (While\ b\ P),\ snd\ (last\ ((Some\ P,\ s)\ \#\ xs)))\ \#\ ys$
$\in assum\ (pre,\ rely)$

From $\models P\ sat\ [pre \cap b,\ rely,\ guar,\ pre]$ the precondition (which plays the role of the loop-invariant) holds after the full execution of the body. It is also easy to prove that if all environment transitions in a computation satisfy the rely condition, so do those of a subcomputation.

Soundness of the Rule of Consequence

The proof of the soundness of the consequence rule is trivial.

lemma *Conseq-sound*:
$[\![pre \subseteq pre';\ rely \subseteq rely';\ guar' \subseteq guar;\ post' \subseteq post;$
$\models P\ sat\ [pre',\ rely',\ guar',\ post']\ ]\!] \Longrightarrow \models P\ sat\ [pre,\ rely,\ guar,\ post]$

The soundness of the system follows from the soundness of each rule. This concludes the proof of the theorem *rgsound*.

The next step is to prove soundness of the system for deriving correct parallel programs. This result relies on an important property of the semantics which we show in the following section.

4.5.2 Compositionality of the Semantics

The most important virtue of the semantics presented in §4.2, where we distinguish between component and environment transitions, is that it allows us to define computations of parallel programs in terms of the computations of the components. In this sense, we say that the semantics is compositional.

A computation c of a parallel program can be described in terms of a list of computations of component programs *clist* if they *conjoin*, and we write it $c \propto clist$.

Before giving the formal definition of the conjoin-operator, we explain its intuitive meaning by means of a parallel program consisting of two components P and Q that run in parallel with an overall environment R. Then, P's environment consists of Q and R. Analogously, Q's environment consists of P and R.

If P and Q are executed in parallel, their respective computations should have the same sequence of states:

$$\begin{array}{ll}(P_0,\ \sigma_0) \xrightarrow{\delta_1} (P_1,\ \sigma_1) \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_n} (P_n,\ \sigma_n) \xrightarrow{\delta_{n+1}} \ldots, & \delta_i \in \{e,\ c\} \\ (Q_0,\ \sigma_0) \xrightarrow{\delta'_1} (Q_1,\ \sigma_1) \xrightarrow{\delta'_2} \ldots \xrightarrow{\delta'_n} (Q_n,\ \sigma_n) \xrightarrow{\delta'_{n+1}} \ldots, & \delta'_i \in \{e,\ c\}\end{array}$$

All components of a parallel composition have computations of the same length. If one component terminates before the others, its computation is extended by environment transitions, which are also allowed when the program has terminated.

Another requirement for separate computations of components to make part of a parallel composition is to have compatible simultaneous transitions. This means that they do not have component transitions at the same time, i.e. δ_i and δ'_i cannot be both c.

Moreover, when some component performs a component transition, then the transition of the full parallel composition is also a component transition. The parallel composition executes an environment step only when all components simultaneously perform an environment step.

For example, consider the above transitions with the following labels:

$$\begin{array}{l}(P_0,\ \sigma_0) \xrightarrow{c} (P_1,\ \sigma_1) \xrightarrow{c} \ldots \xrightarrow{e} (P_n,\ \sigma_n) \xrightarrow{e} \ldots \\ (Q_0,\ \sigma_0) \xrightarrow{e} (Q_1,\ \sigma_1) \xrightarrow{e} \ldots \xrightarrow{e} (Q_n,\ \sigma_n) \xrightarrow{e} \ldots.\end{array}$$

Then, both computations could be composed in the following computation of the parallel program formed by P and Q (denoted $P \parallel Q$):

$$(P_0 \parallel Q_0,\ \sigma_0) \xrightarrow{c} (P_1 \parallel Q_1,\ \sigma_1) \xrightarrow{c} \ldots \xrightarrow{e} (P_n \parallel Q_n,\ \sigma_n) \xrightarrow{e} \ldots$$

The formal definitions of the properties involved for a list of computations to conjoin with the computation of a parallel composition are listed below.

constdefs
same-length :: α *par-confs* $\Rightarrow$ α *confs list* $\Rightarrow$ *bool*
same-length c clist $\equiv \forall\, i{<}$*length clist. length* (*clist*!*i*) = *length c*

All computations have the same length and the same state sequence.

$$\begin{array}{l} \textit{same-state} :: \alpha\ \textit{par-confs} \Rightarrow \alpha\ \textit{confs list} \Rightarrow \textit{bool} \\ \textit{same-state}\ c\ \textit{clist} \equiv \forall\, i{<}\textit{length clist}.\ \forall\, j{<}\textit{length}\ c.\ \textit{snd}\ (c!j) = \textit{snd}\ ((\textit{clist}!i)!j) \end{array}$$

The parallel program must at each stage of the computation be formed by combining the program fragments of *clist*:

$$\begin{array}{l} \textit{same-program} :: \alpha\ \textit{par-confs} \Rightarrow \alpha\ \textit{confs list} \Rightarrow \textit{bool} \\ \textit{same-program}\ c\ \textit{clist} \equiv \forall\, j{<}\textit{length}\ c.\ \textit{fst}\ (c!j) = \textit{map}\ (\lambda x.\ \textit{fst}\ (x!j))\ \textit{clist} \end{array}$$

And finally, the labels must be compatible. A transition is labelled as *pc* in the parallel computation if one of the transitions in *clist* at the corresponding position is a *c*-transition, and *pe* if all transitions in *clist* are also made by the environment.

$$\begin{array}{l} \textit{compat-label} :: \alpha\ \textit{par-confs} \Rightarrow \alpha\ \textit{confs list} \Rightarrow \textit{bool} \\ \textit{compat-label}\ c\ \textit{clist} \equiv \forall\, j.\ \textit{Suc}\ j{<}\textit{length}\ c \longrightarrow \\ \quad (c!j\ {-pc}{\rightarrow}\ c!\textit{Suc}\ j \wedge (\exists\, i{<}\textit{length clist}.\ (\textit{clist}!i)!j\ {-c}{\rightarrow}\ (\textit{clist}!i)!\ \textit{Suc}\ j \wedge \\ \qquad\qquad\qquad (\forall\, l{<}\textit{length clist}.\ l{\neq}i \longrightarrow (\textit{clist}!l)!j\ {-e}{\rightarrow}\ (\textit{clist}!l)!\ \textit{Suc}\ j))) \vee \\ \quad (c!j\ {-pe}{\rightarrow}\ c!\textit{Suc}\ j \wedge (\forall\, i{<}\textit{length clist}.\ (\textit{clist}!i)!j\ {-e}{\rightarrow}\ (\textit{clist}!i)!\ \textit{Suc}\ j)) \end{array}$$

A parallel program conjoins with a list of components if the four properties hold:

$$\begin{array}{l} \textit{conjoin} :: \alpha\ \textit{par-confs} \Rightarrow \alpha\ \textit{confs list} \Rightarrow \textit{bool} \quad (\text{-} \propto \text{-}) \\ c \propto \textit{clist} \equiv \textit{same-length}\ c\ \textit{clist} \wedge \textit{same-state}\ c\ \textit{clist} \wedge \\ \qquad\qquad \textit{same-program}\ c\ \textit{clist} \wedge \textit{compat-label}\ c\ \textit{clist} \end{array}$$

We now prove a lemma stating that the set of computations of a (non-empty) parallel program consists of the computations that conjoin with some list of computations of the components.

theorem *one*: $xs{\neq}[] \Longrightarrow$

$$\begin{array}{l} \textit{par-cp}\ xs\ s = \{c.\ \exists\, \textit{clist}.\ \textit{length clist} = \textit{length}\ xs \wedge c \propto \textit{clist} \\ \qquad\qquad \wedge (\forall\, i{<}\textit{length clist}.\ (\textit{clist}!i) \in \textit{cp}\ (xs!i)\ s)\} \end{array}$$

Hence, the computation of a parallel program can be described in terms of the computations of its components, revealing the compositionality of the semantics[1].

[1]The "numbered" names of some lemmas follow the terminology in [Xu *et al.*, 1997].

Proof. The if-implication

lemma *one-if*:
$[\![$ *length clist* $=$ *length xs*; $\forall\, i$<*length clist*. (*clist*!*i*) $\in$ *cp* (*xs*!*i*) *s*; *c* $\propto$ *clist* $]\!]$
$\Longrightarrow$ *c* $\in$ *par-cp xs s*

is proved by means of the following auxiliary (and equivalent) lemma:

lemma *aux-if*:
$[\![$ *length clist* $=$ *length xs*; $\forall\, i$<*length xs*. (*xs*!*i*, *s*) # *clist*!*i* $\in$ *cptn*;
(*xs*, *s*) # *ys* $\propto$ *map* (λ*i*. (*fst i*, *s*) # *snd i*) (*zip xs clist*) $]\!]$
$\Longrightarrow$ (*xs*, *s*) # *ys* $\in$ *par-cptn*

The proof of *aux-if* proceeds by induction on the list *ys*. The base case is solved by the *ParCptnOne* rule. The induction step is reduced by working out the first step distinguishing whether it is a step made by the environment or by the component. Then we can use the induction hypothesis on the rest of the list.

The only-if-direction is analogously proven by means of an auxiliary lemma:

lemma *aux-onlyif*:
(*xs*, *s*) # *ys* $\in$ *par-cptn* $\Longrightarrow$ $\exists$ *clist*. *length clist* $=$ *length xs* $\wedge$
(*xs*, *s*) # *ys* $\propto$ *map* (λ*i*. (*fst i*, *s*) # *snd i*) (*zip xs clist*) $\wedge$
($\forall\, i$ < *length xs*. (*xs*!*i*, *s*) # (*clist*!*i*) $\in$ *cptn*)

The proof is by induction on *ys*. The base case is solved by instantiating *clist* with the empty list. The proof of the inductive step follows by case analysis on the inductive definition of *par-cptn*, and instantiating *clist* with the appropriate list in each case.

Finally, the equivalence lemma

lemma *one-iff-aux*: *xs*$\neq$[] $\Longrightarrow$
($\forall$ *ys*. (*xs*, *s*) # *ys* $\in$ *par-cptn* $=$
($\exists$ *clist*. *length clist* $=$ *length xs* $\wedge$
(*xs*, *s*) # *ys* $\propto$ *map* (λ*i*. (*fst i*, *s*) # *snd i*) (*zip xs clist*) $\wedge$
($\forall\, i$<*length xs*. (*xs*!*i*, *s*) # *clist*!*i* $\in$ *cptn*))) $=$
(*par-cp* (*xs*) *s* $=$
{*c*. $\exists$ *clist*. *length clist* $=$ *length xs* $\wedge$ *c* $\propto$ *clist* $\wedge$
($\forall\, i$<*length clist*. *clist*!*i* $\in$ *cp* (*xs*!*i*) *s*)})

proves that theorem *one* follows from the two auxiliary lemmas. □

The compositionality of the semantics is necessary for the proof of soundness of the rule for parallel programs, subject of the next section.

4.5.3 Soundness of the System for Parallel Programs

This section is devoted to the soundness of the system of rules that define the set *par-rghoare*. The type of c in a derivable formula $\vdash c\ SAT\ [pre, rely, guar, post]$ is a list of the complete specifications of the program components. However, for the validity formula we just require the corresponding parallel program. We obtain it from c with the function

constdefs
$\quad ParallelCom :: \alpha\ rgformula\ list \Rightarrow \alpha\ par\text{-}com$
$\quad ParallelCom\ Ps \equiv map\ (Some \circ fst)\ Ps$

The soundness theorem is formulated using this function as follows:

theorem *par-rgsound*:
$\quad \vdash c\ SAT\ [pre, rely, guar, post] \Longrightarrow$
$\quad\ \models ParallelCom\ c\ SAT\ [pre, rely, guar, post]$

Proof. The proof proceeds by rule induction. The system *par-rghoare* consists of a single rule: *Parallel*. The soundness of the system is thus reduced to the soundness proof of this rule, namely

lemma *Parallel-sound*:
$\quad [\![\forall i{<}length\ xs.\ rely \cup (\bigcup j{\in}\{j.\ j{<}length\ xs \wedge j{\neq}i\}.\ Guar\ (xs!j)) \subseteq Rely\ (xs!i);$
$\quad\ (\bigcup j{\in}\{j.\ j{<}length\ xs\}.\ Guar\ (xs!j)) \subseteq guar;$
$\quad\ pre \subseteq (\bigcap i{\in}\{i.\ i{<}length\ xs\}.\ Pre\ (xs!i));$
$\quad\ (\bigcap i{\in}\{i.\ i{<}length\ xs\}.\ Post\ (xs!i)) \subseteq post;$
$\quad \forall i{<}length\ xs.\ \models Com(xs!i)\ sat\ [Pre(xs!i), Rely(xs!i), Guar(xs!i), Post(xs!i)]\ ]\!]$
$\quad \Longrightarrow\ \models ParallelCom\ xs\ SAT\ [pre, rely, guar, post]$

By the soundness of the system for component programs we can assume that all component programs are valid. Our proof follows the one presented in [Xu *et al.*, 1997]. It relies on the compositionality of the semantics and four other lemmas that need this property for their proofs.

The second lemma states the following: Given the assumptions of the lemma *Parallel-sound*, if a computation x of a parallel program satisfies the assumptions and conjoins with a list of computations of component programs *clist*, then all component transitions in each of the component computations satisfy their corresponding guarantee conditions.

lemma *two*:

$$\begin{array}{l}
[\![\forall\, i{<}length\ xs.\ rely \cup (\bigcup j{\in}\{j.\ j{<}length\ xs \wedge j{\neq}i\}.\ Guar\ (xs!j)) \subseteq Rely\ (xs!i);\\
\quad pre \subseteq (\bigcap i{\in}\{i.\ i < length\ xs\}.\ Pre\ (xs!i));\ \ length\ xs = length\ clist;\\
\ \forall\, i{<}length\ xs.\ \models Com\ (xs!i)\ sat\ [Pre\ (xs!i),\ Rely\ (xs!i),\ Guar\ (xs!i),\ Post\ (xs!i)];\\
\ x \in par\text{-}cp\ (ParallelCom\ xs)\ s;\ \ x \in par\text{-}assum\ (pre,\ rely);\\
\ \forall\, i{<}length\ clist.\ clist!i \in cp\ (Some\ (Com\ (xs!i)))\ s;\ \ x \propto clist\ ]\!]\\
\ \Longrightarrow \forall\, i{<}length\ clist.\ \forall\, j.\ Suc\ j{<}length\ x \longrightarrow clist!i!j\ {-c\rightarrow}\ clist!i!Suc\ j \longrightarrow\\
\qquad\qquad (snd\ (clist!i!j),\ snd\ (clist!i!Suc\ j)) \in Guar\ (xs!i)
\end{array}$$

The proof proceeds by contradiction. Assume that the first c-transition which does not satisfy the guarantee condition is from $xs!i$ at step m. From the compositionality of the semantics, each e-transition in the subcomputation $take\ (Suc\ (Suc\ m))\ (clist!i)$ corresponds to a c-transition in one of the other components or to an e-transition of x, therefore it satisfies $rely \cup (\bigcup j{\in}\{j.\ j < length\ xs \wedge j \neq i\}.\ Guar\ (xs!j))$. Hence, we can prove

$$take\ (Suc\ (Suc\ m))\ (clist!i) \in assum\ (Pre\ (xs!i),\ Rely\ (xs!i))$$

But this contradicts

$$\models Com\ (xs!i)\ sat\ [Pre\ (xs!i),\ Rely\ (xs!i),\ Guar\ (xs!i),\ Post\ (xs!i)]$$

because one c-transition within the first m transitions does not satisfy the guarantee condition.

Given the assumptions of the previous lemma, the third lemma states that all e-transitions of each of the component computations $xs!i$ satisfy

$$rely \cup (\bigcup j{\in}\{j.\ j < length\ xs \wedge j \neq i\}.\ Guar\ (xs!j)).$$

lemma *three*:

$$\begin{array}{l}
[\![\ xs \neq [];\ pre \subseteq (\bigcap i{\in}\{i.\ i < length\ xs\}.\ Pre\ (xs!i));\ \ length\ xs = length\ clist;\\
\ \forall\, i{<}length\ xs.\ rely \cup (\bigcup j{\in}\{j.\ j < length\ xs \wedge j \neq i\}.\ Guar\ (xs!j)) \subseteq Rely\ (xs!i);\\
\ \forall\, i{<}length\ xs.\ \models Com\ (xs!i)\ sat\ [Pre\ (xs!i),\ Rely\ (xs!i),\ Guar\ (xs!i),\ Post\ (xs!i)];\\
\ x \in par\text{-}cp\ (ParallelCom\ xs)\ s;\ x \in par\text{-}assum\ (pre,\ rely);\\
\ \forall\, i{<}length\ clist.\ clist!i \in cp\ (Some\ (Com\ (xs!i)))\ s;\ x \propto clist\ ]\!]\\
\ \Longrightarrow \forall\, i{<}length\ clist.\ \forall\, j.\ Suc\ j{<}length\ x \longrightarrow clist!i!j\ {-e\rightarrow}\ clist!i!Suc\ j
\end{array}$$

$\longrightarrow (snd\ (clist!i!j),\ snd\ (clist!i!Suc\ j)) \in$
$rely \cup (\bigcup j{\in}\{j.\ j < length\ xs \wedge j \neq i\}.\ Guar\ (xs!j))$

The proof follows directly from lemma *two*.

The forth lemma says that each *pc*-transition satisfies *guar*:

lemma *four*:
$[\![\ xs{\neq}[];\ x \in par\text{-}cp\ (ParallelCom\ xs)\ s;\ x \in par\text{-}assum\ (pre,\ rely);$
$\forall i{<}length\ xs.\ rely \cup (\bigcup j{\in}\{j.\ j{<}length\ xs \wedge j{\neq}i\}.\ Guar\ (xs!j)) \subseteq Rely\ (xs!i);$
$(\bigcup j{\in}\{j.\ j < length\ xs\}.\ Guar\ (xs!j)) \subseteq guar;$
$pre \subseteq (\bigcap i{\in}\{i.\ i < length\ xs\}.\ Pre\ (xs!i));$
$\forall i{<}length\ xs.\ \models Com\ (xs!i)\ sat\ [Pre\ (xs!i),\ Rely\ (xs!i),\ Guar\ (xs!i),\ Post\ (xs!i)];$
$Suc\ i < length\ x;\ x!i\ {-pc\rightarrow}\ x!Suc\ i\ ]\!]$
$\Longrightarrow (snd\ (x!i),\ snd\ (x!Suc\ i)) \in guar$

The proof follows from lemma *two*.

The last lemma states that if the computation terminates, the final state satisfies all postconditions of the components and thus the postcondition of the parallel program:

lemma *five*:
$[\![\ xs{\neq}[];\ x \in par\text{-}cp\ (ParallelCom\ xs)\ s;\ x \in par\text{-}assum\ (pre,\ rely);$
$\forall i{<}length\ xs.\ rely \cup (\bigcup j{\in}\{j.\ j{<}length\ xs \wedge j{\neq}i\}.\ Guar\ (xs!j)) \subseteq Rely\ (xs!i);$
$\forall i{<}length\ xs.\ \models Com\ (xs!i)\ sat\ [Pre\ (xs!i),\ Rely\ (xs!i),\ Guar\ (xs!i),\ Post\ (xs!i)];$
$pre \subseteq (\bigcap i{\in}\{i.\ i < length\ xs\}.\ Pre\ (xs!i));$
$(\bigcap i{\in}\{i.\ i < length\ xs\}.\ Post\ (xs!i)) \subseteq post;$
$fst\ (last\ x) = ys;\ \ All\text{-}None\ ys\ ]\!]\ \Longrightarrow snd\ (last\ x) \in post$

From lemma *one* there exists a list of computations *clist* such that $x \propto clist$. From lemma *three* and the hypothesis it follows that

$$\forall i{<}length\ clist.\ clist!i \in assum\ (Pre\ (xs!i),\ Rely\ (xs!i))$$

By validity of the component programs the last state of each one satisfies $Post\ (xs!i)$. From the definition of conjoining computations the last state of all component computations is the same as the last state of x. Thus, it satisfies *post*.

The soundness of the parallel composition rule *Parallel-sound* follows directly from lemmas *four* and *five*. This concludes the proof of *par-rgsound*. □

The soundness proofs of both systems represent the theoretical part of the formalization of the rely-guarantee system. The next two sections are concerned with the practical application. Section 4.6 defines concrete syntax for programs in order to facilitate verification of real programs and section 4.7 presents some examples.

4.6 Concrete Syntax

This section presents an alternative external representation for programs and assertions. It approaches the standard syntax used in the literature providing a user-friendlier interface for the application of the formalized rely-guarantee method on concrete programs.

The concrete syntax defined here is similar to that of §2.7 where we presented the concrete syntax for the language of the Owicki-Gries formalization. In particular, the representation of program variables follows the quote/antiquote technique. We refer to section 2.7.1 for detailed explanations. Here, we concentrate on the particularities of the concrete syntax for the language of the present chapter. An overview of the different elements and their external representation is shown in table 4.1.

In a rely-guarantee specification the pre- and postcondition are sets of single states. Like in 2.7 variables within such assertions are represented in sans serif font, e.g. $\{\!|\mathsf{x} = 0|\!\}$. Internally, x is a selector function that refers to the value of the variable x at some state.

The rely and guarantee conditions are, however, sets of pairs of states. We need to refer to the values of the variables of the first and second states, i.e. the values before and after the transition. In this case, using simply a selector function is not enough. Antiquoted entities refer now to a pair. The solution is to "antiquote" the selector function that refers to a variable, e.g. x, composed with the predefined functions on pairs *fst* and *snd*, for the value of the variable x before or after the transition, respectively. For example, the set $\{(s_1, s_2) \mid s_1\ s_2.\ x\ s_1 = 0 \wedge x\ s_2 = 1\}$ is represented by the expression $\{\!|\,\acute{}(x \circ \mathit{fst}) = 0 \wedge\ \acute{}(x \circ \mathit{snd}) = 1|\!\}$ or equivalently $\{\!|\ x\ \acute{}\mathit{fst} = 0 \wedge x\ \acute{}\mathit{snd} = 1\ |\!\}$, in the concrete syntax. These expressions are, however, quite long, so we introduce new syntax for antiquotations referring to the *before* and *after* the transition.

syntax
  -before $:: (\alpha \Rightarrow \beta) \Rightarrow \beta$ $({}^{\mathrm{o}}\text{-})$
  -after $:: (\alpha \Rightarrow \beta) \Rightarrow \beta$ $({}^{\mathrm{a}}\text{-})$

Assertion	$\{\!| r |\!\}$
Sequential commands	
Basic	$\mathsf{x} := e$
Seq	$c_0;;\ c_1$
Cond	**if** b **then** c_0 **else** c_1 **fi**
While	**while** b **do** c **od**
Await	**await** b **then** c **end**
Skip	**skip**
Cond2	**if** b **then** c_0 **fi**
Atom	$\langle c \rangle$
Wait	**wait** b **end**
Parallel commands	
Parallel	**cobegin** $s_0 \parallel \ldots \parallel s_n$ **coend**
Schematic	**scheme** $[j \leq i < k]\ s$
Notation	
x: program variable	
$\overline{\mathsf{x}}$: value of program variable after a transition	
e: expression of the type of x	
r, b: boolean expressions	
c, c_0, c_1: sequential commands	
s, s_0, $\ldots$ s_n: specifications for component programs	
j, k: limits for indexing parameterized programs	

Table 4.1: Concrete syntax for programs.

Note that these syntax declarations are similar to that of the syntax constant *-quote* of §2.7.1. The corresponding translations into internal syntax are:

translations
$^{\mathrm{o}}x \rightharpoonup x\ \acute{}fst$
$^{\mathrm{a}}x \rightharpoonup x\ \acute{}snd$

In the examples shown in this thesis we tune the output by printing entities preceded by an antiquote symbol $\acute{}$ in sans serif font. To maintain this nice output for all variables we print entities preceded by $^{\mathrm{o}}$ also in sans serif and those preceded by $^{\mathrm{a}}$ are printed in sans serif and overlined, e.g. the set $\{\!|\ ^{\mathrm{o}}x = {}^{\mathrm{a}}x\ |\!\}$ is printed $\{\!|\ \mathsf{x} = \overline{\mathsf{x}}\ |\!\}$.

The declaration of syntax constants, the equational and the parse and print translations that translate the external concrete syntax into the internal abstract syntax and vice versa are very similar to those defined for the language of the Owicki-Gries formalization (see appendix B).

4.7 Examples

We show the application of the method on three known examples from the literature [Xu *et al.*, 1997, Stirling, 1988].

4.7.1 Set Elements of an Array to Zero

The first example is a very simple program. It sets the first n components of an array A to zero in parallel. There are no shared variables and no auxiliary variables are required for the verification.

record *Example*1 =
  A :: *nat list*

lemma *Example*1:
$\vdash$ **cobegin**
    scheme $[0 \leq i < n]$
   $(\mathsf{A} := \mathsf{A}\ [i := 0],$
   $\{\!| \ n < \mathit{length}\ \mathsf{A} \ |\!\},$
   $\{\!| \ \mathit{length}\ \mathsf{A} = \mathit{length}\ \overline{\mathsf{A}} \wedge \mathsf{A}\ !\ i = \overline{\mathsf{A}}\ !\ i \ |\!\},$
   $\{\!| \ \mathit{length}\ \mathsf{A} = \mathit{length}\ \overline{\mathsf{A}} \wedge (\forall j{<}n.\ i \neq j \longrightarrow \mathsf{A}\ !\ j = \overline{\mathsf{A}}\ !\ j) \ |\!\},$
   $\{\!| \ \mathsf{A}\ !\ i = 0 \ |\!\})$
  coend
 $\mathit{SAT}\ [\{\!| \ n < \mathit{length}\ \mathsf{A} \ |\!\}, \{\!| \ \mathsf{A} = \overline{\mathsf{A}} \ |\!\}, \{\!| \ \mathit{True} \ |\!\}, \{\!| \ \forall i < n.\ \mathsf{A}\ !\ i = 0 \ |\!\}]$

The array (modeled as a list) must have at least as many elements as the number of parallel components; this is the only requirement in the precondition. The rely condition of component i requires that the environment does not change the value of the array A at index i. On the other hand, it guarantees not to change the values of the other indices. The length of the array is invariant, thus this holds in both the rely and the guarantee condition.

We consider this program as closed with respect to the environment. This is reflected in the overall rely condition which requires that the environment does not affect the variables used in the program. The overall guarantee condition can be set to *True*, because no other program relies on the behavior of this one. If there was an influencing environment, its effect could be stated in the overall rely and guarantee conditions

The proof just requires us to first apply the *Parallel* rule backwards. The rule *Basic* is used for the proof of derivability of the parameterized component program. The generated verification conditions are easily proven with standard Isabelle techniques.

4.7.2 Increment a Variable in Parallel

Consider the program $x := x{+}1 \parallel x := x{+}1$. The goal is to prove that if the precondition $x = 0$ holds, then the postcondition $x = 2$ is satisfied by the final states. This parallel program is a classic in the literature because, despite its simplicity, auxiliary variables are unavoidable for its verification.

We declare the shared variable x and two "private" auxiliary variables c_0 and c_1, one for each component:

record *Example*2 =
  x :: *nat*
  c_0 :: *nat*
  c_1 :: *nat*

The program must be extended with assignments to the auxiliary variables. The first and second components satisfy complementary specifications, where the rely and guarantee conditions are switched. We verify the parallel composition as a closed system:

lemma *Example*2:
$\vdash$ **cobegin**
$(\langle \mathsf{x} := \mathsf{x}{+}1;;\ \mathsf{c}_0 := \mathsf{c}_0 + 1\rangle,$
$\{\!| \mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \wedge \mathsf{c}_0 = 0 |\!\},$
$\{\!| \mathsf{c}_0 = \overline{\mathsf{c}_0} \wedge (\mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \longrightarrow \overline{\mathsf{x}} = \overline{\mathsf{c}_0} + \overline{\mathsf{c}_1}) |\!\},$
$\{\!| \mathsf{c}_1 = \overline{\mathsf{c}_1} \wedge (\mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \longrightarrow \overline{\mathsf{x}} = \overline{\mathsf{c}_0} + \overline{\mathsf{c}_1}) |\!\},$
$\{\!| \mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \wedge \mathsf{c}_0 = 1 |\!\})$
$\parallel$
$(\langle \mathsf{x} := \mathsf{x}{+}1;;\ \mathsf{c}_1 := \mathsf{c}_1{+}1\rangle,$

$\{\!\!\{ \mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \wedge \mathsf{c}_1 = 0 \ \}\!\!\}$,
$\{\!\!\{ \mathsf{c}_1 = \overline{\mathsf{c}_1} \wedge (\mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \longrightarrow \overline{\mathsf{x}} = \overline{\mathsf{c}_0} + \overline{\mathsf{c}_1}) \}\!\!\}$,
$\{\!\!\{ \mathsf{c}_0 = \overline{\mathsf{c}_0} \wedge (\mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \longrightarrow \overline{\mathsf{x}} = \overline{\mathsf{c}_0} + \overline{\mathsf{c}_1}) \}\!\!\}$,
$\{\!\!\{ \mathsf{x} = \mathsf{c}_0 + \mathsf{c}_1 \wedge \mathsf{c}_1 = 1 \}\!\!\})$
coend
SAT $[\{\!\!\{ \mathsf{x} = 0 \wedge \mathsf{c}_0 = 0 \wedge \mathsf{c}_1 = 0 \}\!\!\}, \{\!\!\{ \mathsf{x} = \overline{\mathsf{x}} \wedge \mathsf{c}_0 = \overline{\mathsf{c}_0} \wedge \mathsf{c}_1 = \overline{\mathsf{c}_1} \}\!\!\}$,
$\{\!\!\{ \mathit{True} \}\!\!\}, \{\!\!\{ \mathsf{x} = 2 \}\!\!\}]$

The proof uses the rule for parallel composition *Parallel* and the system of rules for sequential component programs.

Parameterized Version

We take advantage of the possibility of proving the derivability of parameterized programs in our system and show in the next lemma how to prove the correctness of the last example for any number of components.

We introduce a composed variable c, such that the component i of the parallel program atomically updates the shared variable x and the ith component of c.

There are several possibilities of modeling such composed variables in HOL. One of them is as lists of length n (like in the previous example), however, lists cause in some cases unnecessary trouble, for example the invariance of the length has to be explicitly required in all assertions. Another more abstract possibility is to use functions from naturals to the value domain of the components. The syntax for updating the value of a function f on argument i is $f\ (i := t)$, where t is of the corresponding type.

The declaration of the variables is:

record *Example*2*-parameterized* =
x :: *nat*
c :: *nat* $\Rightarrow$ *nat*

We want to prove that the extended parameterized program

$$\langle \mathsf{x} := \mathsf{x}{+}1,, \mathsf{c} := \mathsf{c}\ (i := 1)\rangle$$

satisfies the rely-guarantee specification

$$(\mathsf{x} = 0 \wedge (\textstyle\sum i < n.\ \mathsf{c}\ i) = 0, \mathsf{x} = \overline{\mathsf{x}} \wedge \mathsf{c} = \overline{\mathsf{c}},\ \mathit{True}, \mathsf{x} = n).$$

For the assertions, we use the summation function predefined in the Isabelle library. In the precondition of the program we require the summation of the values of the composed variable c on the first n natural numbers to be 0. The rely and guarantee conditions indicate that the program is to be executed in a closed environment.

To establish the specification above, we require that each component i satisfy suitable (parameterized) local specifications.

The lemma stating the derivability of the full specification is:

lemma *Example2-parameterized*: $0 < n \Longrightarrow$
$\vdash$ **cobegin**
  scheme $[0 \leq i < n]$
  $(\langle \mathsf{x} := \mathsf{x}{+}1;; \mathsf{c} := \mathsf{c}\ (i := 1)\rangle,$
  $\{\!|\ \mathsf{x} = (\sum i < n.\ \mathsf{c}\ i) \wedge \mathsf{c}\ i = 0\ |\!\},$
  $\{\!|\ \mathsf{c}\ i = \bar{\mathsf{c}}\ i \wedge (\mathsf{x} = (\sum i < n.\ \mathsf{c}\ i) \longrightarrow \bar{\mathsf{x}} = (\sum i < n.\ \bar{\mathsf{c}}\ i))\ |\!\},$
  $\{\!|\ (\forall j < n.\ i \neq j \longrightarrow \mathsf{c}\ j = \bar{\mathsf{c}}\ j) \wedge$
   $(\mathsf{x} = (\sum i < n.\ \mathsf{c}\ i) \longrightarrow \bar{\mathsf{x}} = (\sum i < n.\ \bar{\mathsf{c}}\ i))\ |\!\},$
  $\{\!|\ \mathsf{x} = (\sum i < n.\ \mathsf{c}\ i) \wedge \mathsf{c}\ i = 1\ |\!\})$
 coend
$SAT\ [\{\!|\ \mathsf{x} = 0 \wedge (\sum i < n.\ \mathsf{c}\ i) = 0\ |\!\}, \{\!|\ \mathsf{x} = \bar{\mathsf{x}} \wedge \mathsf{c} = \bar{\mathsf{c}}\ |\!\}, \{\!|\ True\ |\!\}, \{\!|\ \mathsf{x} = n\ |\!\}]$

It is fairly easy to prove with the help of some basic lemmas about the summation function.

In section 5.5 of the next chapter we reconsider this example and study the relation with its proof in the Owicki-Gries system.

4.7.3 Find Least Element

The next example was already used in [Owicki and Gries, 1976a] and has ever since constituted a standard example of parallel program verification. We reproduce the more general version for parameterized parallel programs of [Stirling, 1988].

Let B be an array, modeled as a list of length m. We want a program that finds the first element, if there is one, satisfying the predicate P. If so, it is saved in the variable x, otherwise $x = m$. Then, the program *FINDP* should satisfy the specification

$$\{\!|True|\!\}\ FINDP\ \{\!|x < m{+}1 \wedge (\forall i < x.\ \neg\ P\ (B!i)) \wedge x < m \longrightarrow P\ (B!x)|\!\}$$

where we already use the notation of Isabelle for access to components of a list. The program *FINDP* is of the form: *INIT*; *SEARCH*; *END*, where *INIT* initializes, *SEARCH* searches in parallel and *END* performs the final assignment to x.

In *SEARCH* we use a number of concurrent programs, *SEARCH* (0) $\parallel \ldots \parallel$ *SEARCH* $(n{-}1)$ with $n \leq m$; for simplicity let n divide m. Each *SEARCH* (i) scans the array squares i, $n{+}i$, $2{*}n{+}i$, $\ldots$ $m{+}i$ looking for x.

Assume that each *SEARCH* (i) has a private variable x_i for searching. Each *SEARCH* (i) should terminate when

1. $P(B \; ! \; x_i)$ or
2. $x_i > m$ or
3. *SEARCH* (k), with $k \neq i$, has found that $P(B \; ! \; x_k)$ for $x_k < x_i$.

We introduce a further private variable y_i for each *SEARCH* (i) which initially is set to $m{+}i$ and if $P(B \; ! \; x_i)$ holds, then y_i is set to x_i. Hence, the termination condition for *SEARCH* (i) is $\exists j < n.\ y_j \leq x_i$. Finally, *END* sets x to the value of $min(y_0, \ldots, y_{n-1})$ when each *SEARCH* (i) terminates.

Consider *SEARCH* (i). Then,

1. As x_i, y_i are private, the environment cannot affect their values or increase y_k, $k \neq i$.
2. The program *SEARCH* (i) cannot affect the variables x_k and y_k for $k \neq i$ and it does not increase the initial value of y_i.

SEARCH (i) will be a loop with invariant:

$$\begin{aligned}\textbf{inv}:\quad & x_i \; mod \; n = i \;\wedge\; (\forall j < x_i.\ j \; mod \; n = i \longrightarrow \neg P(B \; ! \; j)) \;\wedge \\ & (y_i < m \longrightarrow P(B \; ! \; y_i) \;\wedge\; y_i \leq m + i)\end{aligned}$$

The full specification is shown below. Like in the previous example we represent the private variables x_i and y_i as functions x, y from naturals to naturals, so that $x\ i$ corresponds to the private variable x_i of component i:

```
record Example3 =
  x :: nat ⇒ nat
  y :: nat ⇒ nat

lemma Example3: m mod n = 0 ⟹
 ⊢ cobegin
 scheme [0 ≤ i < n]
 (while (∀j < n. x i < y j) do
    if $P\ (B\ !\ \mathsf{x}\ i)$ then $\mathsf{y} := \mathsf{y}\ (i := \mathsf{x}\ i)$
    else $\mathsf{x} := \mathsf{x}\ (i := (\mathsf{x}\ i) + n)$ fi
   od,
  $\{\!|\ (\mathsf{x}\ i)\ mod\ n = i \wedge (\forall j < \mathsf{x}\ i.\ j\ mod\ n = i \longrightarrow \neg P\ (B\ !\ j)) \wedge$
    $(\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ \mathsf{y}\ i) \wedge \mathsf{y}\ i \leq m{+}i)\ |\!\}$,
  $\{\!|\ (\forall j < n.\ i \neq j \longrightarrow \bar{\mathsf{y}}\ j \leq \mathsf{y}\ j) \wedge \mathsf{x}\ i = \bar{\mathsf{x}}\ i \wedge \mathsf{y}\ i = \bar{\mathsf{y}}\ i\ |\!\}$,
  $\{\!|\ (\forall j < n.\ i \neq j \longrightarrow \mathsf{x}\ j = \bar{\mathsf{x}}\ j \wedge \mathsf{y}\ j = \bar{\mathsf{y}}\ j) \wedge \bar{\mathsf{y}}\ i \leq \mathsf{y}\ i\ |\!\}$,
  $\{\!|\ (\mathsf{x}\ i)\ mod\ n = i \wedge (\forall j < \mathsf{x}\ i.\ j\ mod\ n = i \longrightarrow \neg P\ (B\ !\ j)) \wedge$
    $(\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ \mathsf{y}\ i) \wedge \mathsf{y}\ i \leq m{+}i) \wedge (\exists j < n.\ \mathsf{y}\ j \leq \mathsf{x}\ i)\ |\!\}$)
  coend
  $SAT\ [\{\!|\ \forall i < n.\ \mathsf{x}\ i = i \wedge \mathsf{y}\ i = m{+}i\ |\!\}, \{\!|\ \mathsf{x} = \bar{\mathsf{x}} \wedge \mathsf{y} = \bar{\mathsf{y}}\ |\!\}, \{\!|\ True\ |\!\}$,
    $\{\!|\ \forall i < n.\ (\mathsf{x}\ i)\ mod\ n = i \wedge (\forall j < \mathsf{x}\ i.\ j\ mod\ n = i \longrightarrow \neg P\ (B\ !\ j)) \wedge$
     $(\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ \mathsf{y}\ i) \wedge \mathsf{y}\ i \leq m{+}i) \wedge (\exists j < n.\ \mathsf{y}\ j \leq \mathsf{x}\ i)\ |\!\}]$
```

The initialization part of the program (*INIT*) as well as the final assignment $x := min(y_0, \ldots y_{n-1})$ of *END* are not included because the sequential composition with parallel programs is not defined in the programming language. If so wanted, the language should be extended and so the semantics and the proof rules. Here we concentrate on the verification of the parallel part *SEARCH*.

We consider the parallel program closed and thus define the overall rely and guarantee conditions as such.

The interactive proof is easy but needs explicit hints about the assertions that should be used for the final verification conditions. This reveals an important aspect which was not obvious while studying the theory: an automatic verification generation tactic would need intermediate annotations.

It is known that sequential while-programs need only be annotated with the corresponding loop invariants. This is because invariants are not, in general, derivable from the postcondition and the program itself. Loop invariants, precondition and postcondition are the only annotations that an automatic procedure requires to extract the verification conditions for a sequential program.

For the Owicki-Gries method, more annotations than loop-invariants were required. In fact, programs had to be fully annotated as proof outlines. The automatic tactic could thus extract the verification conditions out of the intermediate annotations supplied by the user.

In the verification of this example we observe that an automatic tactic to generate the verification conditions is not possible by just annotating programs with the four conditions (pre-, rely-, guar- and postcondition).

While-invariants and even explicit intermediate annotations are also needed. To illustrate the insufficiency of the rely-guarantee specification we follow the verification of this example and show that extra information must be supplied by the user.

An automatic vcg-tactic that uses the specified rules in the theory would proceed in the following order:

1 First, the *Parallel* rule is applied backwards. We obtain four verification conditions and a fifth goal concerning the derivability of the component programs.

2 The component programs are all the same up to indexing, so the goal reduces to the following formula:

```
lemma i < n ⟹
 ⊢ while (∀j < n. x i < y j)
   do if P (B ! x i) then y := y (i := x i)
      else x := x (i := x i + n) fi
   od
 sat
 [⦃ x i mod n = i ∧ (∀j < x i. j mod n = i ⟶ ¬ P (B ! j)) ∧
   (y i < m ⟶ P (B ! y i) ∧ y i ≤ m+i) ⦄,
 ⦃ (∀j < n. i ≠ j ⟶ ȳ j ≤ y j) ∧ x i = x̄ i ∧ y i = ȳ i ⦄,
 ⦃ (∀j < n. i ≠ j ⟶ x j = x̄ j ∧ y j = ȳ j) ∧ ȳ i ≤ y i ⦄,
 ⦃ x i mod n = i ∧ (∀j < x i. j mod n = i ⟶ ¬ P (B ! j)) ∧
   (y i < m ⟶ P (B ! y i) ∧ y i ≤ m+i) ∧ (∃j < n. y j ≤ x i) ⦄]
```

3 The precondition in this case has been carefully chosen to be the invariant of the loop, thus we can apply the *While* rule directly. From this rule we obtain five subgoals. Four of them are solvable verification conditions. Only the derivability of the while-body remains:

```
lemma i < n ⟹
 ⊢ if P (B ! x i) then y := y (i := x i)
     else x := x (i := x i + n) fi
  sat
 [⦃ x i mod n = i ∧ (∀j < x i. j mod n = i ⟶ ¬ P (B ! j))
   ∧ (y i < n∗q ⟶ P (B ! y i)) ⦄ ∩ ⦃ ∀j < n. x i <y j ⦄,
```

$\{\!\mid (\forall j < n.\ i \neq j \longrightarrow \bar{\mathsf{y}}\ j \leq \mathsf{y}\ j) \wedge \mathsf{x}\ i = \bar{\mathsf{x}}\ i \wedge \mathsf{y}\ i = \bar{\mathsf{y}}\ i \mid\!\}$,
$\{\!\mid (\forall j < n.\ i \neq j \longrightarrow \mathsf{x}\ j = \bar{\mathsf{x}}\ j \wedge \mathsf{y}\ j = \bar{\mathsf{y}}\ j) \wedge \bar{\mathsf{y}}\ i \leq \mathsf{y}\ i \mid\!\}$,
$\{\!\mid \mathsf{x}\ i\ mod\ n = i \wedge (\forall j < \mathsf{x}\ i.\ j\ mod\ n = i \longrightarrow \neg\ P\ (B\ !\ j)) \wedge$
$(\mathsf{y}\ i < n{*}q \longrightarrow P\ (B\ !\ \mathsf{y}\ i)) \mid\!\}$]

The problem we observe is that the precondition of this subgoal is automatically *pre* $\cap$ *b* (see rule *Cond*), which is the strongest precondition. When we attempt to verify the derivability of this subprogram, we fail at condition *stable pre rely* which results from applying the rule *Cond*.

The rely condition only ensures that the values of $x\ i$ and $y\ i$ remain invariant for the component i, but we cannot prove that $\forall j < n.\ x\ i < y\ j$, because the environment can decrease the values of some $y\ k$ with $k \neq i$.

However, we can verify the program if we first apply the *Conseq* rule and choose a weaker precondition, namely, the same one but where the condition b only refers to the variables with index i, i.e. $x\ i < y\ i$. Obviously *pre* $\cap$ *b* implies the weaker precondition. If we apply the rule *Cond* backwards with this new weaker precondition we can prove *stable pre rely*.

For an automatic tactic it is not possible to guess when the intermediate assertions should be weakened or strengthened. Even more difficult would be to guess the new pre-, or postconditions that yield a successful derivation. The solution lies in defining a new set of inference rules for annotated programs and designing the automatic tactic in terms of these rules.

4.8 Concluding Remarks

The rely-guarantee method represents the logical successor of Owicki-Gries as its compositional reformulation. We have formalized the system and a soundness proof following the presentation given in [Xu *et al.*, 1997]. Often, the definitions and proofs found in the literature turn out to be too inefficient for theorem proving techniques. In this formalization, for example, we give an alternative (but equivalent) definition for the semantics in order to carry out the proofs successfully.

So far, we have applied the formalization to the verification of simple programs. The verifications have been done by interactively applying the proof rules and using the standard Isabelle techniques for proving the generated verification conditions. This works fine for small programs, but otherwise it becomes quite tedious. To handle the verification of larger programs a tactic for the automatic generation of the verification conditions should be designed. Such a tactic requires information about intermediate annotations, which means that programs have to be presented as proof outlines. Another interesting extension is the inclusion of nested parallelism in the language. This would allow us to use the compositionality of the method for top-down design and verification of large systems. Both extensions involve changes in the definition of the programming language, which would influence the whole formalization. However, the proof of soundness is essentially the same as the one presented in [Xu *et al.*, 1997] where nested parallelism is part of the language. Thus, the formalized proofs would only need minor modifications.

Chapter 5

Completeness of the Proof Systems for Parameterized Parallel Programs

In the previous chapters we have formalized the Owicki-Gries and the rely-guarantee systems in Isabelle/HOL and proved soundness of these systems w.r.t. the corresponding underlying semantics.

The complementary property of soundness is completeness. A system for verification of programs is *complete* if all correct programs can be deduced in the system. Soundness is a minimal requirement of such systems. One starts with a trivial sound system, for example, the empty one, and adds sound axioms and rules which preserve soundness until a complete system is achieved.

Our current proof systems are incomplete because there is no rule for removing auxiliary variables. This rule is fundamental for a completeness proof. However, for the practical purposes of this work, i.e. mechanical verification of parallel programs, it is sufficient to find derivations for programs augmented with assignments to auxiliary variables. When these variables are used correctly they influence neither control nor data flow. Consequently, removing all assignments to auxiliary variables does not affect the correctness of the program.

The systems with the rule for removing auxiliary variables have been proved complete in previous works. Completeness proofs for the Owicki-Gries system can be found in [Owicki, 1975], [Apt, 1981a] or in [de Roever *et al.*, 2000]. For the rely-guarantee system see for example [Xu *et al.*, 1997], [Stirling, 1988] or again [de Roever *et al.*, 2000]. Both systems are proven to be relatively complete with respect to the standard interpretation in the natural numbers. Moreover, in [Apt, 1981a] the author proves that the assertions for the Owicki-Gries system are recursive when history variables are used as auxiliary variables but only recursively enumerable when program counters are used.

We assume these results to be known and concentrate on extending the completeness property to a kind of programs not considered in the completeness proofs mentioned above, namely, parallel programs where the number of parallel components is represented by a parameter n. Contrary to the traditional systems found in the literature, these programs can be directly derived in our systems. The reason for this lies in the representation of parallel programs as lists of component programs.

The chapter is organized as follows. In section 5.1 we give a formal characterization of parameterized programs. Section 5.2 presents the completeness proof of the extended Owicki-Gries system. Section 5.4 extends this result to the rely-guarantee method. In section 5.5 we illustrate by an example the relevance of these completeness results.

The results presented in this chapter are, apart from the examples of section 5.5, not carried out with Isabelle and thus independent from the details of the formalization. In the sequel, we use a standard notation (even for program syntax) which is not necessarily the one required by the theorem prover.

5.1 Parameterized Programs

Parameterized programs represent a family of programs by a single syntactic object. In order to represent, understand and manipulate these objects, several aspects of the programming language have to be adapted to deal with parameters. In this section we give a formal description of the syntax of parameterized programs and other aspects relevant for the following sections.

5.1.1 Syntax of Parameterized Programs

Let n and i be parameters, where n ranges over the natural numbers and i ranges over the subset $\{0, \ldots, n-1\}$. A *parameterized component program* $S(i,\ n)$, parameterized by n and i, is described by the following syntactic sets:

1. A finite set of simple variables $V = \{x_0, \ldots, x_k\}$, where we assume without loss of generality that each x_j for $j \leq k$ ranges over the natural numbers, and a finite set of composed variables $V_c = \{y_0, \ldots, y_l\}$. (Simple variables could also be considered a degenerated case of composed variables.) Composed variables are usually implemented as arrays, where the value of an array is a function from some "range" to some "set" of values [Best, 1996]. Access to components outside the range of an array are not allowed. We interpret composed variables directly on $I\!N$ in a way that we explain in §5.2.1.

2. Parameterized arithmetic and boolean expressions given by the following BNF grammars:

$$\begin{aligned}
&a ::= j \mid m \mid N \mid x \mid y[a_0] \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1\\
&b ::= \mathit{True} \mid b_0 \wedge b_1 \mid \neg b_0 \mid \exists x.\ b_0 \mid a_0 \leq a_1
\end{aligned}$$

   where j and m are special variables that take the values given by the parameters i and n, N is a natural number, $x \in V$ and $y \in V_c$.

   Given an arithmetic expression a, we define $a(i,\ n)$ inductively as follows:

$$\begin{aligned}
&j(i,\ n) \overset{\text{def}}{=} i, \quad m(i,\ n) \overset{\text{def}}{=} n, \quad N(i,\ n) \overset{\text{def}}{=} N,\\
&x(i,\ n) \overset{\text{def}}{=} x, \quad y[a_0](i,\ n) \overset{\text{def}}{=} y[a_0(i,\ n)],\\
&(a_0 + a_1)(i,\ n) \overset{\text{def}}{=} a_0(i,\ n) + a_1(i,\ n),\\
&(a_0 - a_1)(i,\ n) \overset{\text{def}}{=} a_0(i,\ n) - a_1(i,\ n),\\
&(a_0 * a_1)(i,\ n) \overset{\text{def}}{=} a_0(i,\ n) * a_1(i,\ n)
\end{aligned}$$

   Parameterized boolean expressions $b(i,\ n)$ are defined analogously.

3. Finally, the syntax of parameterized component programs:

$$\begin{aligned}
S ::= \quad & x := a \mid y[a_0] := a_1 \mid S_1;\ S_2 \mid\\
& \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid\\
& \textbf{while } b \textbf{ do } S \textbf{ od} \mid \textbf{await } b \textbf{ then } T \textbf{ end}
\end{aligned}$$

We define $S(i,\ n)$ inductively as follows:

$$(x := a)(i,\ n) \overset{\text{def}}{=} x := a(i,\ n),$$
$$(y[a_0] := a_1)(i,\ n) \overset{\text{def}}{=} y[a_0(i,\ n)] := a_1(i,\ n),$$
$$(S_1;\ S_2)(i,\ n) \overset{\text{def}}{=} S_1(i,\ n);\ S_2(i,\ n),$$
$$(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi})(i,\ n) \overset{\text{def}}{=}$$
$$\textbf{if } b(i,\ n) \textbf{ then } S_1(i,\ n) \textbf{ else } S_2(i,\ n) \textbf{ fi}$$
$$\ldots$$

The family of parallel programs built form $S(i,\ n)$ is defined by the set $\{\|_{i=0}^{n-1} S(i,\ n) \mid n \in I\!N\}$. The notation $\|_{i=0}^{n-1} S(i,\ n)$ represents the program $S(1,\ n) \| \ldots \| S(n,\ n)$, where $\|$ is the parallel composition construct based on interleaving. For each n, the resulting member is a concrete (non-parameterized) parallel program like the programs considered in the original Owicki-Gries and rely-guarantee systems.

A typical example for a parameterized parallel program is the ticket mutual exclusion algorithm for distributed processes that we already considered in §2.8

$$\textbf{ticket} \equiv \mathit{num} := 1;\ \mathit{next} := 1;\ \mathit{turn} := 0;\ \|_{i=0}^{n-1} S(i,\ n)$$

where $S(i,\ n)$ is shown in Figure 5.1. By the assignment $\mathit{turn} := 0$ we mean that 0 is assigned to each component of the array *turn*.

```
$S(i,\ n) \equiv$  while $\mathit{true}$ do
                     $\mathit{NCS}(i,\ n)$;  (noncritical section)
                     $\langle\ \mathit{turn}[i] := \mathit{num};\ \mathit{num} := \mathit{num} + 1\ \rangle$;
                     wait $\mathit{turn}[i] = \mathit{next}$ end;
                     $\mathit{CS}(i,\ n)$;    (critical section)
                     $\mathit{next} := \mathit{next} + 1$
                   od
```

Figure 5.1: Ticket algorithm.

Syntax of assignments to composed variables

The concrete syntax of the programming language in the Isabelle/HOL formalizations of the previous chapters does not support the above notations for assignments to components of composed variables. For example, an assignment to component i of a list a is written $a := a\ [i{:=}e]$. Similarly, an assigment to the argument i of a function a is written $a := a\ (i{:=}e)$ meaning that the function is assigned the new updated function. In the literature, such assignments are usually written $a[i] := e$, however, both notations have the same semantics, which we explain in the next section.

5.1.2 The State Space

The meaning of a program S is usually defined as a partial function $\mathcal{M}(S)$ from states to states. States are represented as tuples of values corresponding to the variables occurring in the program. For the purpose of our completeness proof it is convenient to require that these values be natural numbers. Thus, given a state s and a simple variable x, we refer to the value of the variable x in state s by $s(x)$. For a composed variable y, $s(y)$ returns the corresponding encoding of the sequence of values given by its components. This means that the value domain of a composed variable is also $I\!N$. As we shall see in §5.3, the reason for this is that free variables in the language of elementary arithmetic must range over numbers and not over functions.

There are many different ways of encoding a finite sequence of values as a single natural number. Any such encoding will do as long as there is an effective procedure for encoding and an effective procedure for extracting any of the original components from the coded sequence. Following the syntax in [Apt, 1981a], the encoding of the sequence $a_1, \ldots, a_k$ is represented by $\lceil a_1, \ldots, a_k \rceil$. If $a = \lceil a_1, \ldots, a_k \rceil$ then $a \frown c = \lceil a_1, \ldots, a_k, c \rceil$. We denote the code of the empty sequence by $\lceil\rceil$. We assume some properties of the functions "$\lceil \ldots \rceil$" and "$\frown$", in particular, their definability in the language of elementary arithmetic. The proofs of these properties can be found in [Schoenfield, 1967]. The state space Σ of a given parameterized program is then $I\!N^k$, where k is the total number of variables appearing in the program.

An assignment to a component i of a composed variable should be understood as an assignment to the composed variable as a whole. Thus, the new value is the composed variable with the component i updated. Due to the special treatment of the value domain for composed variables we informally describe the semantics of assignments to components of composed variables by the following transition rule:

$$(y[a_0] := a_1,\ s) \rightarrow (\mathit{None},\ s[y \leftarrow \mathit{update}(y,\ s(a_0),\ s(a_1))])$$

None denotes the empty program. The notation $s[y \leftarrow t]$ stands for the usual operation of substitution in s of the value of the variable y by the new value t. The syntax $s(a_0)$ denotes the value resulting from the evaluation of the arithmetic expression a_0 by substituting the free variables by the values they have in state s. Finally, $\mathit{update}(y,\ j,\ l)$ is a function that replaces the j^{th} component of the sequence encoded by y with a new value l. It is possible to prove that there is a partial recursive procedure computing this function. Thus, the transformations performed on the state by a program are all computable.

The rules of the semantics for the rest of the constructors can be consulted in sections 2.2 and 4.2.

The next section deals with the completeness result for parameterized parallel programs of the Owicki-Gries system. As will be explained in section 5.4, due to the strong connection between the completeness proofs of the Owicki-Gries and the rely-guarantee systems, and to the independence of the proof in section 5.2 from the details of the Owicki-Gries system, this result can be easily extended to the rely-guarantee system.

5.2 Completeness of the Owicki-Gries System for Parameterized Parallel Programs

The proof rule for parallel composition as formulated in the original Owicki-Gries system is[1]

$$\frac{\begin{array}{c}\vdash \{P_0\}\ S_0\ \{Q_0\}, \ldots, \vdash \{P_k\}\ S_k\ \{Q_k\} \\ \textit{and the proof outlines are interference free}\end{array}}{\vdash \{P_0 \cap \ldots \cap P_k\}\ S_0 \parallel \ldots \parallel S_k\ \{Q_0 \cap \ldots \cap Q_k\}}$$

This rule is concerned with the verification of parallel programs consisting of a fixed number k of (possibly different) components $S_0, \ldots, S_k$. We refer to the Owicki-Gries system by O.

However, as encountered already in the previous chapters, many interesting parallel programs are given schematically in terms of a parameter n, representing the number of parallel components. In order to be able to verify these programs directly we consider the extension of the system O by

[1]Assertions are modeled as sets of states here.

the following rule:

$$\frac{\begin{array}{l}\forall i < n.\ \vdash \{P(i,\ n)\}\ S(i,\ n)\ \{Q(i,\ n)\} \wedge \\ \forall i < n.\ \forall j < n.\ i \neq j \longrightarrow \textit{the proof outlines of} \\ \{P(i,\ n)\}\ S(i,\ n)\ \{Q(i,\ n)\} \wedge\ \{P(j,\ n)\}\ S(j,\ n)\ \{Q(j,\ n)\} \\ \textit{are interference free}\end{array}}{\vdash \{\bigcap_{i=0}^{n-1} P(i,\ n)\}\ \|_{i=0}^{n-1} S(i,\ n)\ \{\bigcap_{i=0}^{n-1} Q(i,\ n)\}}$$

where $P(i,\ n)$ and $Q(i,\ n)$ denote for each i and n the precondition and the postcondition, respectively.

This rule represents a particular case of the more general rule presented in the formalization of the Owicki-Gries system in §2.4.3. Hence, we can easily derive it from the system[2]:

lemma *ParamParallelRule*:
$[\![\ \forall\, i{<}n.\ \vdash (c\ i)\ (Q\ i);$
$\ \ \forall\, i{<}n.\ \forall\, j{<}n.\ i \neq j \longrightarrow \textit{interfree-aux}\ (\textit{Some}\ (c\ i),\ Q\ i,\ \textit{Some}\ (c\ j))\ ]\!]$
$\Longrightarrow \Vdash (\bigcap i \in \{i.\ i < n\}.\ \textit{pre}(c\ i))$
$\quad\quad$ **cobegin scheme** $[0 \leq i < n]\ (c\ i)\ (Q\ i)$ **coend**
$\quad\quad (\bigcap i \in \{i.\ i < n\}.\ Q\ i)$

This rule allows us to prove partial correctness of a parallel program parameterized by the number of component processes n by showing it for an arbitrary but fixed value of n. We refer to the system of rules that results by adding this new rule to the original Owicki-Gries system by F (F stands for *family*). We write $\vdash_F\ \{P\}\ S\ \{Q\}$ to denote that $\{P\}\ S\ \{Q\}$ can be derived from F. The soundness of this system has been already proven in chapter 2. Here we prove the relative completeness of F.

The proof proceeds by induction on the structure of the programs. For all non-parameterized constructors the proofs can be found in, for example [Owicki, 1975]. The only interesting case here is that of parameterized parallel programs, i.e. given a family of valid partial correctness formulas of the form

$$\models \{P(n)\}\ \|_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$$

we want to study if we can always find a proof outline for S so that

$$\vdash_F\ \{P(n)\}\ \|_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$$

[2]This rule is specific for the derivation of one parameterized parallel program. However, the theoretical completeness result presented here can be generalized to the more general rule of the formalization.

By the completeness of the system O we know that for each value of n we can always find proof outlines for each of the n components, so that the program can be derived in O. This result, however, does not give any information about what these proof outlines look like for different values of n and i. Do they have a uniform pattern (probably parameterized by n and i) or are the proof outlines of the program with three components completely different from those of the program with four components? And, for a given value of n, is the annotation of the component i different from the annotation of the component j for some values of i and j in the set $\{0, \ldots, n-1\}$?

Previous works on parallel program verification present proofs of parameterized programs in the style of Owicki and Gries [Stirling, 1988, de Roever *et al.*, 2000]. However, they abstract from the fact that the Owicki-Gries system does not directly support reasoning on parameterized programs. An extension of the system and a proof of completeness was missing. The authors of [de Roever *et al.*, 2000] say

> Since Szymanski's algorithm[3] is parameterized, it is quite natural to use assertion networks parameterized by a process index i.

In the present work we provide a formal proof that supports this "naturality." The main conclusion is that, for any valid specification of a parameterized program, it is always possible to find a single (parameterized) proof outline that can be derived in the system F for all values of n.

The proof is organized as follows. In §5.2.1 we prove the semantic existence of the intermediate assertions that form a proof outline. This means that we show in our meta-language (English and mathematics) that these assertions exist (so-called *semantic completeness*). Further, we prove that these so constructed proof outlines yield valid F-derivations since they satisfy the expected properties for using the method. In §5.3 we show that these (semantic) assertions can be expressed in a language containing at least first order arithmetic over the standard model of the natural numbers (elementary arithmetic). That is, we prove that syntactic expressions corresponding to the semantic ones exist. This result is called *expressiveness*, which together with the semantic completeness implies *relative completeness.*

[3] A mutual exclusion algorithm for distributed processes.

5.2.1 Semantic Completeness

The proof proceeds by induction on the structure of the programs. The case for parallel programs of the form $S_0 \parallel \ldots \parallel S_k$ where the number of components is fixed was first proven in [Owicki, 1975].

The remaining case is that of parameterized parallel programs. Given a valid partial correctness formula of the form

$$\models \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$$

we prove that we can always find a proof outline for S such that

$$\vdash_F \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$$

Given a parameterized (non-annotated) component program $S(i,\ n)$, we construct its proof outline by associating an assertion with every point of interference. For this purpose, it suffices to annotate $S(i,\ n)$ with a postcondition $q(i,\ n)$ and every normal subprogram $R(i,\ n)$ of $S(i,\ n)$ with a precondition $\mathit{pre}(R(i,\ n))$, where a *normal subprogram* of S is a subprogram which is not a proper subprogram of an await-statement. The result is an annotated program $A(S(i,\ n))$. We define $\mathit{pre}(A(S(i,\ n)))$ and $\mathit{post}(A(S(i,\ n)))$ to be the initial precondition and the final postcondition, respectively.

To prove correctness of parameterized parallel programs using the system F, we need to show local correctness and interference freedom of the associated proof outlines. To this end, one must in general augment actions and boolean guards with assignments to auxiliary variables. These record the progress of computation in the other processes allowing us to express the effects of parallel execution.

There are two well-known canonical methods of introducing auxiliary variables (see 2.4.4). One makes use of a single so-called *history* variable z that records, throughout execution, the index number of the component performing the atomic action and the values of the program variables before that action is executed. In a terminating state, z contains the history associated with the executions of assignments or await-statements. The second possibility consists of using auxiliary variables as labels indicating the control points that the program passes through. Thus, these special auxiliary variables (which are also called counters) are updated with *every* transition. One auxiliary variable of this form for each component suffices for any Owicki-Gries style proof [Lamport, 1977, Best, 1996].

Our programming language does not allow for the atomic evaluation of a boolean condition together with the updating of an auxiliary variable. As a consequence, not all transitions, but only assignments and await-statements, can be recorded by auxiliary variables. For this reason we use a single history variable z, respecting this syntactic restriction, in the style of Apt [Apt, 1981a]. The set of program variables is augmented with a single variable z, containing the value corresponding to the encoding of the substring of the history sequence, where transitions corresponding to the evaluation of boolean expressions are not registered.

Given a program S with state space Σ, we denote the state space of the extended program S^* by Σ^*, where we assume that the first value of each tuple corresponds to the value of the history variable. For instance, if $s^* \in \Sigma^*$ is a state of S^*, then, by convention, s denotes the corresponding state of Σ, where the first element of the tuple has been removed.

Given $\models \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$, the proof of semantic completeness is structured as follows:

1. Extend the program by initializing the history variable:

   $$z := [\,];\ \parallel_{i=0}^{n-1} S(i,\ n)$$

2. Extend $S(i,\ n)$ to $S^*(i,\ n)$ by adding assignments to the history variable.

3. Annotate every point of interference in $S^*(i,\ n)$ with an assertion, obtaining an annotated program $A(S^*(i,\ n))$ such that the following conditions hold:

   (a) $\forall s \in \Sigma.\ s \in P(n) \longrightarrow ([\,],\ s) \in \bigcap_{i=0}^{n-1}\ \mathit{pre}(A(S^*(i,\ n)))$

   (b) *Local correctness of proof outlines*: $\forall i < n.\ \vdash A(S^*(i,\ n))$

   (c) *Interference freedom*: $\forall i, j \in \{0, \ldots,\ n-1\}$. If $i \neq j$ then, for every assertion $r_k(i,\ n)$ in $A(S^*(i,\ n))$, and for every atomic action $a(j,\ n)$ with precondition $\mathit{pre}(a(j,\ n))$ in $A(S^*(j,\ n))$, the formula

   $$\vdash \{r_k(i,\ n) \cap \mathit{pre}(a(j,\ n))\}\ a(j,\ n)\ \{r_k(i,\ n)\}$$

   holds.

   (d) $\forall s^* \in \Sigma^*.\ s^* \in \bigcap_{i=0}^{n-1}\ \mathit{post}(A(S^*(i,\ n))) \longrightarrow\ s \in Q(n)$.

Then, using the rule for parameterized parallel programs, we can conclude that $\vdash_F\ \{P(n)\}\ z := [\,];\ \parallel_{i=0}^{n-1} A(S^*(i,\ n))\ \{Q(n)\}$ holds. By deleting the assignments to auxiliary variables with the elimination rule (see 2.4.4) and omitting the intermediate annotations we obtain

$$\vdash_F \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$$

Note that with the system O we could prove the following statement

$$\begin{aligned}&\forall n.\ (\models \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\} \longrightarrow \\ &\exists A^n.\ \vdash_O \{P(n)\} \parallel_{i=0}^{n-1} A^n(S^*(i,\ n))\ \{Q(n)\})\end{aligned}$$

where for different values m and l of n, A^m can be very different from A^l. In this paper, however, we prove the existence of a uniform annotation (parameterized by n and i) for all values of n, i.e. we prove

$$\begin{aligned}&\models \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\} \longrightarrow \\ &\exists A.\ \vdash_F \{P(n)\} \parallel_{i=0}^{n-1} A(S^*(i,\ n))\ \{Q(n)\}\end{aligned}$$

5.2.2 Extending the Program

Let $V = \{x_0, \ldots, x_k\}$ be the set of variables occurring in $S(i,\ n)$. Let $\overline{x}$ denote the coding $\lceil x_0, \ldots, x_k \rceil$. Let z be a new variable. Then, we transform $S(i,\ n)$ into $S^*(i,\ n)$ by replacing

(i) every await-statement **await** b **then** R **end**, where R is not *skip*, by **await** b **then** $z := z \frown \lceil i,\ \overline{x} \rceil;\ R$ **end**,

(ii) every assignment $y := t$ not inside an **await**-statement by **await** *true* **then** $z := z \frown \lceil i,\ \overline{x} \rceil;\ y := t$ **end**.

If the language allows for assignments to variables at every transition (including those consisting of evaluation of boolean conditions), completeness can be proven by extending the program with assignments to counter variables. In a parameterized program the counter variable would be a composed variable c. The ith component program would update this variable by updating its ith component, i.e. $c[i]$, at every transition. As we shall see in §5.5, parallel programs are in practice verified using auxiliary variables as counters because finding assertions in terms of history variables is too difficult.

5.2.3 Annotating the Program

Given $S^*(i,\ n)$, our goal is to construct intermediate assertions[4] that yield a proof outline such that conditions (a), (b), (c) and (d) of §5.2.1 can be proven.

By the completeness of O we know that given

$$\models \{P(n)\} \parallel_{i=0}^{n-1} S(i,\ n)\ \{Q(n)\}$$

we can find for every value m of n, i.e. for the particular valid triple

$$\models \{P(m)\}\ S(1,\ m) \parallel \ldots \parallel S(m,\ m)\ \{Q(m)\},$$

an extended program

$$z := [\,];\ (S^*(1,\ m) \parallel \ldots \parallel S^*(m,\ m))$$

and an annotation A^m so that

$$\vdash \{P(m)\}\ z := [\,];\ \parallel_{i=0}^{m-1} A^m(S^*(i,\ m))\ \{Q(m)\}$$

A^m associates every point of interference j inside a component i with an assertion (set of states) r_{ji}^m whose construction is described in the completeness proof for the system O [Owicki, 1975, Apt, 1981a, de Roever *et al.*, 2000]. Informally, these assertions are defined as the sets of states that are reachable by some computation starting in a state in the precondition, where the steps of the computation are determined by the rules of the operational semantics.

The assertion associated with a location j in a parameterized component program $S^*(i,\ n)$, represented by $r_j(i,\ n)$, is defined as follows:

$$r_j(i,\ n) = r_{ji}^n$$

where for each value m of n and for each value l of i, the set r_{jl}^m is the one constructed in the completeness proof of the Owicki-Gries system for the concrete parallel program $\parallel_{i=0}^{m-1} S(i,\ m)$ at the location j inside component l.

It remains to be proven that with this annotation the conditions necessary to derive the corresponding triple using the system F are satisfied. For this purpose, it suffices to prove that conditions (a), (b), (c) and (d) mentioned above hold for this annotation. This is easy, since for any arbitrary value m of n, the proof is reduced to the case for the concrete program with m parallel components. We just show the proof of (a), the others are similar:

[4]By an assertion we mean here a set of states as a semantic object.

(a) Denote the first location (the one corresponding to the precondition) of a component by 0. Assume that there is a state s such that $s \in P(m)$. By the completeness of the system O we have $(\lceil\rceil, s) \in \bigcap_{i=0}^{m-1} r_{0i}^{m}$. By definition of $r_0(i,\ m)$, we have $(\lceil\rceil, s) \in \bigcap_{i=0}^{m-1} r_0(i,\ m)$.

This closes our proof of semantic completeness of the system F. By defining the intermediate assertions as parameterized functions that for each n and each i return the set of states that characterize the assertions of the particular program with those values of n and i, we have proven that these sets of states exist for every n and every i. The main goal, however, is to prove that these sets not only exist but can also be concretely described using some concrete assertion language.

5.3 Relative Completeness

In this section we prove that the semantic sets described in §5.2.1 can be syntactically expressed if we use an assertion language which contains at least the logical system of elementary arithmetic and consider programs whose boolean conditions and state transformations can be expressed in the assertion language. For this purpose we use an important result of recursion theory [Rogers, 1987]:

> For any relation R, if R defines a recursively enumerable set, then R is definable in elementary arithmetic.

Thus, it suffices to prove that the sets of states $r_j(i,\ n)$ that we constructed in §5.2.1 are recursively enumerable sets. Intuitively, a set S is recursively enumerable (or checkable) if there exists a procedure M such that given an element t as input, "checks" whether t is in S and stops when its checking procedure succeeds. If t is not in S the procedure continues checking forever. We show that such a procedure for checking the sets $r_j(i,\ n)$ exists.

Let $\|_{i=0}^{n-1} S(i,\ n)$ be a parameterized parallel program and let $P(n)$ be a set of initial states. For every value m of n and for every control point j in component i, the set r_{ji}^{m} is recursively enumerable. We denote the corresponding checking procedure by M_{ji}^{m}. Informally, it works as follows:

> Given an input state s, $M_{ji}^{m}(s)$ succeeds iff it is possible to start an execution of $S^*(1,\ m) \parallel \ldots \parallel S^*(m,\ m)$ from a state in $P(m)$ and reach the control point j of component i with the values of the variables as given by the state s.

We now define the procedure M_j that checks if a state s is in $r_j(i,\ n)$:

> It takes as arguments n, i and the state s. With the values of n and i, it builds the concrete program $S^*(1,\ n) \parallel \ldots \parallel S^*(n,\ n)$ and simulates the procedure M^n_{ji} on state s. Then, $M_j(n,\ i,\ s)$ is defined such that it succeeds iff $M^n_{ji}(s)$ succeeds.

Hence, the domain of the procedure M_j (the values for which it succeeds) is a recursively enumerable set. By the theorem mentioned above, the relation that defines this set is definable in elementary arithmetic. Let us recall what this means [Rogers, 1987]:

> An n-ary relation R is *definable in elementary arithmetic* if there is a formula $F\ a_1 \cdots a_n$ with free variables $a_1, \ldots, a_n$ such that
>
> $$R = \{(x_1, \ldots, x_n) \mid F x_1 \cdots x_n\}$$
>
> where, for any integers $x_1, \ldots, x_n$, $F\ x_1 \cdots x_n$ is the result of substituting the numeral expressions of $x_1, \ldots, x_n$ for $a_1, \ldots, a_n$ in $F\ a_1 \cdots a_n$.

The tuples of our set are formed by the values of n, i and those of the program variables that form the state. Consequently, the formula defining this set has as free variables not only the program variables (such as for concrete programs) but also n and i, as intuitively expected.

In the ticket algorithm, for example, the arithmetical expression defining some assertion would have as free variables, n, i, *num, next* and *turn* (all ranging over $I\!N$). Composed variables are considered simple variables with value the encoding of the values of the components. Thus, in the first-order logic expressions of the assertions there are no references to the components of *turn*. However, to reason practically in Hoare style, more expressive logics are used (e.g. HOL). If these logics allow us to express composed variables as functions, then expressions like $turn[i]$ can be written directly.

This completeness proof does not provide an effective methodology for finding the expressions for assertions. The important result is that these expressions always exist. This means that it is always possible to find parameterized proof outlines for valid parameterized parallel programs.

5.4 Completeness of the Rely-Guarantee System for Parameterized Parallel Programs

The completeness proof of the rely-guarantee system for this particular kind of programs follows by the same reasoning as in the previous section. The completeness proof for the standard rely-guarantee system is, like for the Owicki-Gries system, based on the definition of intermediate annotations as strongest postconditions, but where the environment is also taken into account.

Given a rely-guarantee specification $\vdash P\ SAT\ [pre, rely, guar, post]$ for a parallel program P, we associate with every point of interference l in P the corresponding strongest postcondition SP_l. Following [de Roever *et al.*, 2000], a state s belongs to SP_l if there is a computation of P together with its environment that reaches location l of P, starting in a state satisfying *pre*, such that all environment steps satisfy *rely*. Similarly, the so-called strongest guarantee condition SG is defined as the set of pairs of states defining transitions of P which are executed by P in some computation, provided *pre* is satisfied initially, and every environment transition satisfies *rely*.

The proofs of completeness for the rely-guarantee system presented in [Xu *et al.*, 1997] and [de Roever *et al.*, 2000] are essentially based on these constructions. History variables as auxiliary variables are also needed for the proofs.

In [Stirling, 1988], the author presents another proof of completeness for his version of the rely-guarantee system based on a very elegant theory of invariants. The idea is to prove that the rely-guarantee system is complete with respect to the Owicki-Gries system. Intuitively, whenever we have a derivation for a correct program in the Owicki-Gries system, we can find a derivation in the rely-guarantee system by considering the *rely*-condition of a component program P_i to be the predicate describing the set of pairs of states which preserve the annotations of its own proof outline. Thus, no environment transition can falsify the local correctness of the proof outline. For the *guar*-condition it suffices to take the predicate characterizing the set of pairs of states which preserve the annotations in the proof outlines of the other components P_j with $i \neq j$. Basically, a rely-guarantee specification can be obtained from an Owicki-Gries one by registering in the rely and guarantee conditions the information provided by the interference freedom test. We illustrate this idea with a simple example in §5.5.

In any case, proofs of completeness have already been studied in previous works and we are just interested in assuming that given any valid rely-guarantee specification of a parallel program it is possible to construct the subspecifications of its components so that the program can be derived in the system.

The introduction of auxiliary variables for a parameterized parallel program as well as the proof of semantic and relative completeness presented in the previous section for the Owicki-Gries system are independent of the particularities of the Owicki-Gries system itself and can be directly used for the rely-guarantee method.

Consequently, for any valid rely-guarantee specification of a parameterized parallel program it is possible to find a derivation in the rely-guarantee system. Moreover, the assertions needed for it are by construction recursively enumerable and can thus be expressed in elementary arithmetic.

5.5 Example

There are several examples of assertional verification of parameterized parallel programs in the literature [Stirling, 1988, de Roever *et al.*, 2000, Best, 1996]. Some of them have already been studied in the examples of the previous chapters. Here we consider a very simple program that illustrates the practical meaning of the completeness result presented in this chapter.

The example we choose is the classic program $\{x = 0\} \parallel_{i=0}^{n-1} x := x + 1 \; \{x = n\}$. Its verification with the rely-guarantee method has already been studied in §4.7.2. We show here its verification in the Owicki-Gries system and compare both results.

We introduce a composed variable c, such that the component i of the parallel program atomically updates the shared variable x and the ith component of c. We model the composed variable c as a function from naturals to naturals. The syntax for updating the value of a function f on argument i is $f\ (i{:=}t)$, where t is of the corresponding type. The summation function used in the assertions is predefined in the Isabelle library.

record *Schema* =
 x :: *nat*
 c :: *nat* ⇒ *nat*

lemma *Schema*: $0{<}n \Longrightarrow$
$\Vdash$ ⦃x=0 ∧ ($\sum i{<}\ n.$ c i)=0⦄
 cobegin

```
  scheme [$0 \leq i < n$]
$\{\!|$x=($\sum i< n.$ c $i$) $\wedge$ c $i$=0$|\!\}$
 ⟨x:=x+1,, c:=c ($i$:=1)⟩
$\{\!|$x=($\sum i< n.$ c $i$) $\wedge$ c $i$=1$|\!\}$
coend
$\{\!|$x=$n|\!\}$
```

The completeness result for parameterized parallel programs claims that the assertions can be expressed in the language of elementary arithmetic with free variables n, i, x and c. For obvious practical reasons, we prefer to use a higher order language. However, projection and finite summation are both partial recursive functions and as such can be expressed in a first order arithmetic.

As proved in §4.7.2, the correctness of the same program with respect to the rely-guarantee specification

$$(\mathsf{x} = 0 \wedge (\sum i{<}n.\ \mathsf{c}\ i) = 0,\ \mathsf{x} = \bar{\mathsf{x}} \wedge \mathsf{c} = \bar{\mathsf{c}},\ \mathit{True},\ \mathsf{x} = n)$$

where the overall rely and guarantee conditions specify the program as being closed, requires that each component i satisfy the following (parameterized) local specification:

$$\begin{array}{ll}
\textbf{pre}: & \mathsf{x} = \sum i{<}n.\ \mathsf{c}\ i \wedge \mathsf{c}\ i = 0 \\
\textbf{rely}: & \mathsf{c}\ i = \bar{\mathsf{c}}\ i \wedge \\
 & (\mathsf{x} = \sum i{<}n.\ \mathsf{c}\ i \longrightarrow \bar{\mathsf{x}} = \sum i{<}n.\ \bar{\mathsf{c}}\ i) \\
\textbf{guar}: & (\forall j{<}n.\ i \neq j \longrightarrow \ \ \mathsf{c}\ j = \ \bar{\mathsf{c}}\ j) \wedge \\
 & (\mathsf{x} = \sum i{<}n.\ \mathsf{c}\ i \longrightarrow \bar{\mathsf{x}} = \sum i{<}n.\ \bar{\mathsf{c}}\ i) \\
\textbf{post}: & \mathsf{x} = \sum i{<}n.\ \mathsf{c}\ i \wedge \mathsf{c}\ i = 1
\end{array}$$

As observed by [Stirling, 1988], there is a direct connection between the interference freedom tests required in the Owicki-Gries proof of the same program and the rely and guarantee conditions required here.

For component i the interference freedom test is successful if

1. The truth of the equality $x = \sum i< n.\ c\ i$ is preserved and the local variable $c\ i$ is not modified by the actions of the other components.

2. The atomic actions in i do not affect the truth of the assertions in the other components.

The first part is exactly expressed in the rely condition, which represents what the component assumes from the environment. And the second one is reflected on the guarantee condition that represents what the component ensures the environment. In this case, since the program is closed, the environment is represented by the rest of the components.

Observe that for the formulation of the rely and guarantee conditions, no assumption on the concrete implementation of the environment is made. In the Owicki-Gries formalism, however, we need to know the concrete form of the environment to be able to carry out the interference freedom tests.

This example illustrates the intuitive idea that the rely-guarantee method is complete with respect to the Owicki-Gries method, and that this also applies to parameterized programs.

5.6 Summary

The Owicki-Gries and the rely-guarantee systems are the most fundamental assertional methods for reasoning about the correctness of parallel programs with shared variables. However, they were not designed for the verification of parameterized parallel programs, but for programs with a given number of components only. In the previous chapters we have shown that the systems can be generalized to handle parameterized parallel programs and that the resulting systems are sound.

In this chapter we have studied the completeness of the extended systems. The main difficulty, however, lies in a satisfactory formalization of the problem. To this end, we have defined the syntax and meaning of parameterized parallel programs and have formalized what it means for these programs to be verified in a Hoare logic framework.

The proofs and ideas presented in this chapter have been developed in an abstract mathematical style. Hence, they are independent from the particular formalization of the systems in Isabelle. Moreover, the starting point is the assumption that the proof systems found in the literature are complete. This assumption allows us to state that for each instance of a valid specification of a parameterized parallel program there exists a proof in the system. The main result presented in this chapter, namely, the completeness of the extended systems, states that it is always possible to find a *single* proof that works for any number of components. Not surprisingly (but not obviously) the assertions that allow us to derive such proofs are themselves parameterized. Then, the proof for some particular program can be obtained by instantiating the parameters in the assertions.

Chapter 6

Conclusion

This thesis describes the first formalization of the Owicki-Gries and rely-guarantee methods in a theorem prover. Both methods are used to prove partial correctness of parallel imperative programs with shared variables. For each method, the programming language, the operational semantics and the proof rules have been defined in Isabelle/HOL. The proof of soundness of the rules w.r.t. the underlying semantics has also been carried out with the theorem prover. As a result, we obtain a verified framework for proving the correctness of parallel programs with shared variables having the choice between a compositional and a non-compositional axiomatic method.

The theorem prover Isabelle/HOL has provided a user-friendly, reliable and powerful tool that ensures a systematic, correct and understandable development of the theory underlying both verification systems. We have shown that a theorem prover like Isabelle also provides a powerful tool for the application of these methods to real programs. In particular, by considering programs like the parallel garbage collection algorithms we have gone beyond typical “toy” examples. Moreover, since we had no previous (complete) pencil and paper Owicki-Gries proofs for these examples, the tool was useful not only for the a “a posteriori” verification, but also as a checking friend in the search for a proof.

6.1 General Contributions

Apart from this concrete verification environment this thesis provides contributions which are of general interest for the areas of parallel program verification and mechanical formalization of programming languages.

Parameterized Parallel Programs

The formalized systems differ from those found in the literature mainly in that the rule for parallel composition has been generalized to handle parameterized parallel programs. Thus, it is possible to prove correctness of such systems for any instance of the parameter by finding just one derivation. This naturally led to the question whether finding such a derivation was always possible, i.e. whether the systems are complete for these kind of programs.

Although proofs of completeness for the traditional Owicki-Gries and rely-guarantee systems already exist in the literature, none of them considers the question whether parameterized parallel programs can always be verified in a schematic way. From the completeness results of the original systems we know that there is a derivation in the system for each particular instance of the parameter, however, a machine that checks derivations of the systems for all instances would never stop. In contrast, the completeness of our system demonstrates that it is always possible to find a single parameterized derivation of these programs, so that *one* correctness proof is valid for any instance of the parameter.

Concrete Representation of Programs

Isabelle also provides concrete syntax facilities which makes it possible to present programs in a familiar syntax. To achieve this, the main concern is the representation of program variables. In this thesis we have used the *quote/antiquote* technique. This technique is widely used in the functional programming world, but the idea of using it for the purpose of encoding program variables was first proposed by Markus Wenzel in [Wenzel, 2001b]. These techniques represent the most advanced "technology" in the state of the art of program variables representation in HOL: they provide a combination of the advantages of previously used methods without introducing any drawbacks. The examples shown in this dissertation represent the largest application so far and have contributed to reveal many of the advantages of this model.

Automation of the verification process

With the theorem prover we can apply the verification methods systematically ensuring that no mistakes are made and no details ignored. The verification process is made considerably less tedious because the boring and in general easy routine steps involved in such endeavors are taken care of. This allows the user to concentrate on the fundamental aspects of the proof which require intelligent input.

The main disadvantage of using a theorem prover for the verification of programs is that they generally require extensive human guidance, and that this guidance is expressed in terms of the particular theorem prover. This problem can be partially solved by designing a tactic that automatically generates the verification conditions. Such an automatic tactic has been defined for the Owicki-Gries method and it has been successfully applied to the verification of several non-trivial examples. Depending on the complexity of these verifications conditions, proving them could require certain knowledge of the proving techniques. Nevertheless, once the verification conditions are generated and simplified it is easier to see whether they hold or not.

6.2 Statistics of the formalization

We give an overview of the amount of work carried out with Isabelle. Table 6.1 shows the number of specification lines (types, definitions, inductive sets, etc.), the number of lemmas and the total number of proof steps needed for the formalization of the theory underlying the verification methods.

Theory	Spec. lines	Lemmas	Proof steps
Owicki-Gries	220	49	340
Rely-guarantee	330	93	2240
VCG-tactic	190	42	80
Concrete syntax	260	0	0
Total	**1000**	**184**	**2660**

Table 6.1: Statistics of the formalizations.

We observe that the formalization of the rely-guarantee method is more involved and, in particular, the proofs are longer. This is the price we pay in obtaining a compositional method: the underlying theory requires more work, but yields a simpler proof system. The soundness proof only needs to be done once, however, we enjoy the advantages of a compositional system with every verification exercise.

The tactic for the automatic generation of the verification conditions requires a number of rules derived as lemmas from the Owicki-Gries theory. The 190 specification lines correspond to the ML code implementing the tactic.

The concrete syntax for both formalizations involves 260 lines of specification which include the declaration of syntactic constants, the translation equations and the parse and print translation functions programmed in ML. Many of the features are duplicated in both formalizations.

One of the main goals of this work is to demonstrate the applicability of the formalizations on the verification of real programs. We briefly review the results of our experience.

Table 6.2 shows the statistics of the verification of the typical examples done with the Owicki-Gries formalization.

Algorithm	Verif. cond.	Automatic	Lemmas
Peterson	122	122	0
Dijkstra	20	20	0
Ticket	35	24	0
Zero search (with sync.)	98	98	0
Zero search (without sync.)	20	20	0
Producer/Consumer	138	125	3
Total	**433**	**409**	**3**

Table 6.2: Examples verified with Owicki-Gries.

The next table shows the statistics concerning the verification of the two garbage collection (gc) algorithms and the theory of graphs common to both algorithms.

Theory	Spec. lines	Lemmas	Verif. cond	Proof steps
Graph	23	17	-	289
Single-mut. gc	35	28	289	408
Multi-mut. gc	35	36	328	756
Total	**93**	**81**	**617**	**1453**

Table 6.3: Verification of garbage collection algorithms.

Finally, table 6.4 gives an overview of the effort involved in verifying the examples considered for the rely-guarantee method. There is no tactic for the automatic generation of verification conditions, thus the proof steps also contain the number of steps needed in order to apply the rules from the rely-guarantee system.

Algorithm	Verif. cond.	Lemmas	Proof steps
Array to zero	8	3	40
Increment variable	14	3	23
Find least element	22	1	30
Total	**44**	**7**	**93**

Table 6.4: Examples verified with rely-guarantee.

6.3 Further Work

The present work represents a first basic study of verification methods for parallel programs with shared variables in a theorem prover. Several possible extensions from both the mechanical and the theoretical sides of the formalization can be addressed:

1. The completeness proofs for both systems are known theoretical results. It would be interesting to formalize them in the theorem prover especially because our systems also handle the verification of parameterized parallel programs. To obtain a complete system, however, a missing rule in the present formalization, namely the rule for elimination of auxiliary variables, should also be formalized.

2. The programming language considered for both formalizations is fairly simple. The enrichment with nested parallelism would be a meaningful extension, especially for the rely-guarantee system. For the Owicki-Gries method the lack of this feature is not a major disadvantage because all the processes involved in a parallel program have to be known prior to verification. However, the compositionality of the rely-guarantee system makes the method suitable for top-down development of programs. This is important to cope with the design and verification of large programs. Also, to automate the application of the rely-guarantee method, a tactic for the automatic generation of verification conditions similar to the one presented for the Owicki-Gries method should be designed. However, the presentation of programs should be extended to include intermediate annotations in order to avoid the use of the consequence rule at intermediate steps.

3. It would also be interesting to investigate the relation between proofs in both the compositional and non-compositional methods. Especially, to study whether Owicki-Gries proofs can be translated into rely-guarantee proofs following some systematic procedure.

6.4 Experience

By formalizing these methods in Isabelle/HOL we have gained a full understanding of the difficulty involved in designing correct proof methods for the verification of (parallel) programs. The level of detail required for such a formal work naturally leads to approach each step in the formalization with a critical eye, considering first what might come next and studying alternatives that could simplify the formalization. This often leads to improvements over the known definitions and methods found in the literature. However, to understand the theorem proving techniques and learn to optimize the effort involved in any formalization takes a great deal of time, effort and mistakes.

Verifying parallel programs using the methods formalized here can be difficult, but it is mostly tedious work. The interesting part of the verification process is to understand the program and find an intuitive proof of their correctness. However, the projection of this intuition into assertions that have to satisfy a number of requirements (like interference freedom!) implies, in general, changing and tuning the assertions too many times and a great deal of effort is expended in getting details right.

The tool presented here does not directly help in finding the right annotations but at least automates the iterative process of changing assertions and checking the proof again. One thing is sure, the theorem prover will never let the user get away with wrong or incomplete annotations. We believed to have found the perfect annotations many times only to then have the theorem prover reveal subtle (and sometimes obvious) mistakes. From our experience, we do not recommend implicitly trusting a pencil and paper proof correctness for parallel programs.

6.5 To conclude

The main point of this thesis is that, by formalizing (generalized versions) of the two well-known Owicki-Gries and rely-guarantee methods in a theorem prover, it is now possible to obtain mechanized proofs of correctness for general (parameterized and non-parameterized) parallel programs with these methods.

Appendix A

Automatic Generation of Verification Conditions

We construct a tactic for the automatic generation of the verification conditions (vcg) for full specifications of parallel programs. The construction is presented in a bottom-up style. First, a vcg-tactic is designed for atomic programs. This makes understanding the construction easier, since atomic programs are the simplest ones used here. Using this tactic another vcg-tactic is developed for annotated programs. This second vcg-tactic is employed in the construction of the final vcg-tactic, which is used to generate the verification conditions of parallel programs.

A.1 VCG for Atomic Programs

Atomic programs are sequential programs and do not contain intermediate annotations. The vcg-tactic works by deriving all intermediate assertions from the postcondition and loop invariants and establishing the corresponding verification conditions. These are expressed simply as set theoretic inclusions because assertions and boolean conditions are formalized as sets of states.

The generated intermediate assertions are generally called *weakest liberal preconditions* [Winskel, 1993, Dijkstra, 1976]. Informally, given a command c and its postcondition q, the weakest liberal precondition $wlp(c,\ q)$ is defined as the set of states from which the execution of c either diverges, or ends up in a final state satisfying q.

Given a program and its specification, the verification conditions are obtained by recursively applying the proof rules backwards until only statements of the logic of assertions, without any mention of the programming language, remain. To this end, the proof rules have to be generalized because the rules defined in the proof theory cannot always be applied backwards directly. The pre- and postconditions in most of the conclusions are too specific and need to be previously manipulated via the rule of consequence. Consider for example the following specification of an atomic program:

lemma $\Vdash \{x.\ x = 0\}\ \mathit{Basic}\ (\lambda x.\ x{+}1)\ \{x.\ x = 1\}$

The rule *Basic* is too specific to be applied. The consequence rule comes to the rescue

apply (*rule Conseq*)

generating the following three subgoals:

1. $\{x.\ x = 0\} \subseteq\ ?p$
2. $\Vdash\ ?p\ \mathit{Basic}\ (\lambda x.\ x + 1)\ ?q$
3. $?q \subseteq \{x.\ x = 1\}$

where *?p* and *?q* are schematic variables that can be instantiated with anything of the corresponding type. By applying the rule *Basic* backwards on the second subgoal, the schematic variables *?p* and *?q* are automatically instantiated with the pre- and postcondition required by the rule:

prefer 2
apply (*rule Basic*)

Two verification conditions are left, namely

1. $\{x.\ x = 0\} \subseteq \{s.\ s + 1 \in\ ?q\}$
2. $?q \subseteq \{x.\ x = 1\}$

which are automatically solved by the tactic *auto*:

apply *auto*
done

In general, it is impossible for an automatic tactic to guess when and how many times the rule of consequence should be used to obtain suitable assertions. Instead, we derive more general rules by combining the original proof rules for each command with the rule of consequence:

lemma *SkipRule*: $p \subseteq q \Longrightarrow \Vdash p\ (Basic\ id)\ q$
lemma *BasicRule*: $p \subseteq \{s.\ (f\ s) \in q\} \Longrightarrow \Vdash p\ (Basic\ f)\ q$
lemma *SeqRule*: $[\![\Vdash p\ c_1\ r; \Vdash r\ c_2\ q\ ]\!] \Longrightarrow \Vdash p\ (Seq\ c_1\ c_2)\ q$
lemma *CondRule*:
$[\![\ p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}; \Vdash w\ c_1\ q; \Vdash w'\ c_2\ q\ ]\!]$
$\Longrightarrow \Vdash p\ (Cond\ b\ c_1\ c_2)\ q$
lemma *WhileRule*: $[\![\ p \subseteq i; \Vdash (i \cap b)\ c\ i\ ; (i \cap -b) \subseteq q\ ]\!]$
$\Longrightarrow \Vdash p\ (While\ b\ i\ c)\ q$

These derived rules can always be applied. Only the rule for sequential composition remains the same. Maybe surprising is the case of the conditional statement because the rule of consequence affects the premises and not the conclusion. The original rule

$$[\![\Vdash (p \cap b)\ c_1\ q; \Vdash (p \cap -b)\ c_2\ q\ ]\!] \Longrightarrow \Vdash p\ (Cond\ b\ c_1\ c_2)\ q$$

matches any pre- and postcondition, however, the generated subgoals easily fail. For example, consider the triple

$$p\ (Cond\ True\ (Basic\ id)\ (Basic\ id))\ p.$$

Although it is trivially valid, the original rule *Cond* would fail.

In [Winskel, 1993], this problem is solved by generating the verification conditions from programs that have been previously annotated, not only with loop invariants, but also with suitable assertions before any if- or while-statement. Our approach, however, reduces the required human guidance to loop invariants.

To verify sequential programs we proceed "bottom-up", i.e. starting by the last non-sequential command. Hence, a function called *WlpTac* recursively decomposes sequential constructions[1] until it reaches the last non-sequential command whose postcondition is known. Using the derived proof rules, the tactic generates the weakest (liberal) precondition.

The tactic controls via the parameter *precond* if the precondition is unknown. If so, the generated verification condition has the form $?p \subseteq \ldots$, where $?p$ represents the unknown precondition which is subsequently instantiated by reflexivity, i.e. using the theorem $A \subseteq A$. Initially, the only known preconditions are those supplied by the user, namely the overall precondition and loop invariants. Thus, when the tactic is invoked *precond* is true; it becomes false whenever the precondition must be worked out from the postcondition.

[1]The sequential operator associates to the right, i.e. $c_0;;\ (c_1;;\ c_2)$.

The vcg-tactic for atomic programs is called *HoareRuleTac* and is programmed in ML. The tacticals *THEN*, *ORELSE*, *EVERY* and *FIRST* provide control structures for combining tactics. *THEN* executes one after the other and *ORELSE* tries first one and, if it fails, it tries the second one. *EVERY* and *FIRST* are the corresponding block versions of *THEN* and *ORELSE*, respectively. The tactic *rtac* takes two arguments, a theorem and a (subgoal) number, then it applies the given theorem backwards to the subgoal at the given number. To access Isabelle theorems inside an **ML** environment, they have to be preceded by the word *thm*.

```
ML {*
fun WlpTac i = rtac (thm SeqRule) i THEN HoareRuleTac false (i+1)
and HoareRuleTac precond i =
      WlpTac i THEN HoareRuleTac precond i
    ORELSE
      FIRST[rtac (thm SkipRule) i,
            rtac (thm BasicRule) i,
            EVERY[rtac (thm CondRule) i,
                  HoareRuleTac false (i+2),
                  HoareRuleTac false (i+1)],
            EVERY[rtac (thm WhileRule) i,
                  HoareRuleTac true (i+1)]]
       THEN (if precond then (K all-tac i) else rtac (thm subset-refl) i)
*}
```

The tactic (*K all-tac*) leaves the subgoal unchanged.

A.2 VCG for Component Programs

The vcg-tactic for component programs is simpler because the rules of the system are already generic enough to be directly applied. Moreover, since all annotations must be provided in advance there is no need for finding weakest preconditions.

We only derive three new proof rules for special instances of the *AnnBasic* and the *AnnAwait* commands when the transformation performed on the state is the identity,

lemma *AnnSkipRule*: $r \subseteq q \Longrightarrow \vdash (AnnBasic\ r\ id)\ q$
lemma *AnnWaitRule*: $[\![\ r \cap b \subseteq q\ ]\!] \Longrightarrow \vdash (AnnAwait\ r\ b\ (Basic\ id))\ q$

and for an *AnnAwait* command where the boolean condition is $\{s.\ True\}$:

lemma *AnnAtomRule*:
  $[\![$ *atom-com* c; $\Vdash r\ c\ q$ $]\!] \Longrightarrow \vdash$ (*AnnAwait* r $\{s.\ True\}$ c) q

To refer to a rule of the proof system, for example *AnnBasic*, we have to write *oghoare-ann-hoare.AnnBasic*. This is quite long, so we introduce abbreviated names with the keyword *lemmas*:

lemmas *AnnBasic* = *oghoare-ann-hoare.AnnBasic*
lemmas *AnnSeq* = *oghoare-ann-hoare.AnnSeq*
lemmas $AnnCond_1$ = *oghoare-ann-hoare.*$AnnCond_1$
lemmas $AnnCond_2$ = *oghoare-ann-hoare.*$AnnCond_2$
lemmas *AnnWhile* = *oghoare-ann-hoare.AnnWhile*
lemmas *AnnAwait* = *oghoare-ann-hoare.AnnAwait*
lemmas *AnnConseq* = *oghoare-ann-hoare.AnnConseq*

The vcg-tactic is otherwise similar to the previous one.

```
ML {*
fun AnnWlpTac i = rtac (thm AnnSeq) i THEN AnnHoareRuleTac (i+1)
and AnnHoareRuleTac i =
      AnnWlpTac i THEN AnnHoareRuleTac i
    ORELSE
      FIRST[rtac (thm AnnSkipRule) i,
              EVERY[rtac (thm AnnAtomRule) i,
                      HoareRuleTac true (i+1)],
              rtac (thm AnnWaitRule) i,
              rtac (thm AnnBasic) i,
              EVERY[rtac (thm AnnCond1) i,
                      AnnHoareRuleTac (i+3),
                      AnnHoareRuleTac (i+1)],
              EVERY[rtac (thm AnnCond2) i,
                      AnnHoareRuleTac (i+1)],
              EVERY[rtac (thm AnnWhile) i,
                      AnnHoareRuleTac (i+2)],
              EVERY[rtac (thm AnnAwait) i,
                      HoareRuleTac true (i+1)]]
*}
```

Although it is not necessary to generate the intermediate preconditions from the last postcondition, the tactic also processes the commands “bottom-up”, i.e. the higher subgoal number first. This is necessary because, in general, the application of the proof rules backwards generates more subgoals.

Let us see what happens if the tactic starts by processing the lowest subgoal number. We illustrate the problem by an example,

lemma
$\vdash$ *AnnSeq*
  ($AnnCond_1$ $\{x.\ x = 0\}$ $\{x.\ x = 0\}$
    (*AnnBasic* $\{x.\ x = 0\}$ $(\lambda x.\ x{+}1)$)
    (*AnnBasic* $\{x.\ x = 0\}$ $(\lambda x.\ 0)$))
  (*AnnBasic* $\{x.\ x = 1\}$ $(\lambda x.\ x{+}2)$)
$\{x.\ x = Suc\ 2\}$

First, we decompose the sequential composition,

apply(*rule AnnSeq*)

1. $\vdash AnnCond_1\ \{x.\ x = 0\}\ \{x.\ x = 0\}\ (AnnBasic\ \{x.\ x = 0\}\ (\lambda x.\ x + 1))$
   $(AnnBasic\ \{x.\ x = 0\}\ (\lambda x.\ 0))\ pre\ (AnnBasic\ \{x.\ x = 1\}\ (\lambda x.\ x + 2))$
2. $\vdash AnnBasic\ \{x.\ x = 1\}\ (\lambda x.\ x + 2)\ \{x.\ x = Suc\ 2\}$

If we start by the first command of the sequential composition and apply the rule $AnnCond_1$, the second part of the program is moved 3 subgoals down:

apply(*rule* $AnnCond_1$)

1. $\{x.\ x = 0\} \cap \{x.\ x = 0\} \subseteq pre\ (AnnBasic\ \{x.\ x = 0\}\ (\lambda x.\ x + 1))$
2. $\vdash AnnBasic\ \{x.\ x = 0\}\ (\lambda x.\ x + 1)\ pre\ (AnnBasic\ \{x.\ x = 1\}\ (\lambda x.\ x + 2))$
3. $\{x.\ x = 0\} \cap -\ \{x.\ x = 0\} \subseteq pre\ (AnnBasic\ \{x.\ x = 0\}\ (\lambda x.\ 0))$
4. $\vdash AnnBasic\ \{x.\ x = 0\}\ (\lambda x.\ 0)\ pre\ (AnnBasic\ \{x.\ x = 1\}\ (\lambda x.\ x + 2))$
5. $\vdash AnnBasic\ \{x.\ x = 1\}\ (\lambda x.\ x + 2)\ \{x.\ x = Suc\ 2\}$

In general it is difficult to keep track of the number of subgoals that are generated. However, if we start processing the subgoals "bottom-up", the subgoals with lower numbers remain where they are, no matter how many new subgoals are generated.

A.3 VCG for Parallel Programs

For the constructors shared between atomic and parallel programs the vcg-tactic is exactly like *HoareRuleTac*. We only have to add to this tactic the case where a command is a parallel composition.

First, we derive a rule where the pre- and postcondition of the *Parallel* rule are generalized:

lemma *ParallelConseqRule*:
$\llbracket\ p \subseteq (\bigcap i \in \{i.\ i < \mathit{length}\ Ts\}.\ \mathit{pre}\ (\mathit{the}\ (\mathit{com}\ (Ts!i))));$
$\Vdash (\bigcap i \in \{i.\ i < \mathit{length}\ Ts\}.\ \mathit{pre}\ (\mathit{the}\ (\mathit{com}\ (Ts!i))))$
$(\mathit{Parallel}\ Ts)$
$(\bigcap i \in \{i.\ i < \mathit{length}\ Ts\}.\ \mathit{post}\ (Ts!i));$
$(\bigcap i \in \{i.\ i < \mathit{length}\ Ts\}.\ \mathit{post}\ (Ts!i)) \subseteq q\ \rrbracket \Longrightarrow \Vdash p\ (\mathit{Parallel}\ Ts)\ q$

Then, the *Parallel* rule should be applied backwards on the subgoal that results from the second premise of this rule. As a result, two new subgoals, namely, the derivability of the component programs and their interference freedom, are generated. However, the first subgoal, which results from the first premise of the rule for parallel composition *Parallel*, is quite complicated because of the universal quantifier. We derive a variant of the rule *Parallel* that is more suitable for backwards application.

The predicate $[\vdash]\ Ts$ indicates if all elements in the list of specifications of component programs *Ts* are derivable:

constdefs *map-ann-hoare* :: $(\alpha\ \mathit{ann\text{-}com\text{-}op} \times \alpha\ \mathit{assn})\ \mathit{list} \Rightarrow \mathit{bool}$ $([\vdash]\ \text{-})$
$[\vdash]\ Ts \equiv \forall i < \mathit{length}\ Ts.\ \exists c\ q.\ Ts!i = (\mathit{Some}\ c,\ q) \wedge \vdash c\ q$

This definition corresponds to the first premise of the *Parallel* rule. We substitute it in original proof rule:

lemma *ParallelRule*: $\llbracket\ [\vdash]\ Ts;\ \mathit{interfree}\ Ts\ \rrbracket$
$\Longrightarrow \Vdash (\bigcap i \in \{i.\ i < \mathit{length}\ Ts\}.\ \mathit{pre}\ (\mathit{the}\ (\mathit{com}\ (Ts!i))))$
$\mathit{Parallel}\ Ts$
$(\bigcap i \in \{i.\ i < \mathit{length}\ Ts\}.\ \mathit{post}\ (Ts!i))$

To automate the proof of $[\vdash]\ Ts$ we derive two rules that distinguish whether the list is empty or not, and whether it is a parameterized list of component programs:

lemma *MapAnnEmpty*: $[\vdash]\ []$
lemma *MapAnnList*: $\llbracket \vdash c\ q;\ [\vdash]\ xs\ \rrbracket \Longrightarrow [\vdash]\ (\mathit{Some}\ c,\ q)\#xs$
lemma *MapAnnMap*:
$\forall k.\ a \le k \wedge k < b \longrightarrow \vdash (c\ k)\ (Q\ k) \Longrightarrow [\vdash]\ \mathit{map}\ (\lambda k.\ (\mathit{Some}\ (c\ k),\ Q\ k))\ [a..b(]$

By using these rules we avoid dealing with the quantifier of the original *Parallel* rule. Observe that for the case of parameterized programs, if we apply *MapAnnMap* backwards, eliminate the quantifier in the premise and "move" $a \leq k \wedge k < b$ to the assumptions, we obtain the subgoal $a \leq k \wedge k < b \Longrightarrow \vdash c\ k\ Q\ k$ for an arbitrary but fixed value of k. Consequently, proving the derivability of a parameterized list of programs is reduced to proving derivability of an arbitrary but fixed component without need for induction.

Proving the predicate *interfree Ts* can also be optimized by new derived rules. First, we define a function such that given a component program x and a list of component programs xs, it checks if the assertions in x are preserved by all atomic actions in xs and if all assertions in xs are preserved by the atomic actions in x:

constdefs $\textit{interfree-swap} :: (\alpha\ \textit{ann-triple-op} \times \alpha\ \textit{ann-triple-op list}) \Rightarrow \textit{bool}$
$\quad \textit{interfree-swap} \equiv \lambda(x,\ xs).\ \forall y \in \textit{set}\ xs.\ \textit{interfree-aux}\ (\textit{com}\ x,\ \textit{post}\ x,\ \textit{com}\ y)$
$\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \textit{interfree-aux}\ (\textit{com}\ y,\ \textit{post}\ y,\ \textit{com}\ x)$

With this function we derive proof rules for *interfree*:

lemma *interfree-Empty*: *interfree* []
lemma *interfree-List*:
$\quad [\![\ \textit{interfree-swap}\ (x,\ xs);\ \textit{interfree}\ xs\ ]\!] \Longrightarrow \textit{interfree}\ (x\#xs)$
lemma *interfree-Map*:
$\quad \forall i\ j.\ a \leq i \wedge i < b \wedge a \leq j \wedge j < b \wedge i \neq j \longrightarrow \textit{interfree-aux}\ (c\ i,\ Q\ i,\ c\ j)$
$\qquad \Longrightarrow \textit{interfree}\ (\textit{map}\ (\lambda k.\ (c\ k,\ Q\ k))\ [a..b(])$

The definitions of *interfree-swap* and *interfree-aux* can also be expressed as inference rules suitable for automation. For *interfree-swap* we derive the following three rules:

lemma *interfree-swap-Empty*: *interfree-swap* (*x*, [])
lemma *interfree-swap-List*:
$\quad [\![\ \textit{interfree-aux}\ (\textit{com}\ x,\ \textit{post}\ x,\ \textit{com}\ y);\ \textit{interfree-aux}\ (\textit{com}\ y,\ \textit{post}\ y\ ,\ \textit{com}\ x);$
$\quad\ \textit{interfree-swap}\ (x,\ xs)\ ]\!] \Longrightarrow \textit{interfree-swap}\ (x,\ y\#xs)$
lemma *interfree-swap-Map*:
$\quad \forall k.\ a \leq k \wedge k < b \longrightarrow \textit{interfree-aux}\ (\textit{com}\ x,\ \textit{post}\ x,\ c\ k)$
$\quad\ \wedge\ \textit{interfree-aux}\ (c\ k,\ Q\ k,\ \textit{com}\ x)$
$\quad \Longrightarrow \textit{interfree-swap}\ (x,\ \textit{map}\ (\lambda k.\ (c\ k,\ Q\ k))\ [a..b(])$

And for *interfree-aux* the next three ones:

lemma *interfree-aux-rule*1: *interfree-aux* (*co*, *q*, *None*)
lemma *interfree-aux-rule*2:

$\forall (R, r) \in (\mathit{atomics}\ a).\ \Vdash (q \cap R)\ r\ q \Longrightarrow \mathit{interfree\text{-}aux}\ (\mathit{None},\ q,\ \mathit{Some}\ a)$

lemma *interfree-aux-rule*3:

$\forall (R, r) \in (\mathit{atomics}\ a).\ \Vdash (q \cap R)\ r\ q \wedge (\forall p \in (\mathit{assertions}\ c).\ \Vdash (p \cap R)\ r\ p)$
$\Longrightarrow \mathit{interfree\text{-}aux}\ (\mathit{Some}\ c,\ q,\ \mathit{Some}\ a)$

The premises of the last two rules are, due to the universal quantifiers, not yet suitable for automation. The solution lies in proving a separate rule for all possible instances of the commands c and a in the conclusion. For each case, the assertions and atomic commands involved are known. The idea is to first extract the assertions that have to be checked for interference and then the atomic commands that should preserve those assertions. In the end, we obtain a subgoal for each of the Hoare triples involved in the interference freedom test.

The rules for isolating the assertions are:

lemma *AnnBasic-assertions*:

$[\![\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ r,\ \mathit{Some}\ a);\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ q,\ \mathit{Some}\ a)\ ]\!] \Longrightarrow$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ (\mathit{AnnBasic}\ r\ f),\ q,\ \mathit{Some}\ a)$

lemma *AnnSeq-assertions*:

$[\![\ \mathit{interfree\text{-}aux}\ (\mathit{Some}\ c_1,\ q,\ \mathit{Some}\ a);\ \mathit{interfree\text{-}aux}\ (\mathit{Some}\ c_2,\ q,\ \mathit{Some}\ a)\ ]\!] \Longrightarrow$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ (\mathit{AnnSeq}\ c_1\ c_2),\ q,\ \mathit{Some}\ a)$

lemma *AnnCond*$_1$*-assertions*:

$[\![\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ r,\ \mathit{Some}\ a);\ \mathit{interfree\text{-}aux}\ (\mathit{Some}\ c_1,\ q,\ \mathit{Some}\ a);$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ c_2,\ q,\ \mathit{Some}\ a)\ ]\!] \Longrightarrow$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ (\mathit{AnnCond}_1\ r\ b\ c_1\ c_2),\ q,\ \mathit{Some}\ a)$

lemma *AnnCond*$_2$*-assertions*:

$[\![\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ r,\ \mathit{Some}\ a);\ \mathit{interfree\text{-}aux}\ (\mathit{Some}\ c,\ q,\ \mathit{Some}\ a)\ ]\!] \Longrightarrow$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ (\mathit{AnnCond}_2\ r\ b\ c),\ q,\ \mathit{Some}\ a)$

lemma *AnnWhile-assertions*:

$[\![\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ r,\ \mathit{Some}\ a);\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ i,\ \mathit{Some}\ a);$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ c,\ q,\ \mathit{Some}\ a)\ ]\!] \Longrightarrow$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ (\mathit{AnnWhile}\ r\ b\ i\ c),\ q,\ \mathit{Some}\ a)$

lemma *AnnAwait-assertions*:

$[\![\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ r,\ \mathit{Some}\ a);\ \mathit{interfree\text{-}aux}\ (\mathit{None},\ q,\ \mathit{Some}\ a)\ ]\!] \Longrightarrow$
$\mathit{interfree\text{-}aux}\ (\mathit{Some}\ (\mathit{AnnAwait}\ r\ b\ c),\ q,\ \mathit{Some}\ a)$

By repeatedly applying these rules backwards only subgoals of the form *interfree-aux* (*None*, r, *Some* a) are left. That is, we eliminate quantification over assertions. Finally, the assertion r has to be checked for invariance under all atomic actions of a. This quantifier is "unfolded" by using rules that distinguish on the command a:

lemma *AnnBasic-atomics*:
$\Vdash (q \cap r)\ (Basic\ f)\ q \Longrightarrow interfree\text{-}aux\ (None,\ q,\ Some\ (AnnBasic\ r\ f))$
lemma *AnnSeq-atomics*:
$[\![\ interfree\text{-}aux\ (c,\ q,\ Some\ a1);\ interfree\text{-}aux\ (c,\ q,\ Some\ a2)\ ]\!] \Longrightarrow$
$interfree\text{-}aux\ (c,\ q,\ Some\ (AnnSeq\ a1\ a2))$
lemma *AnnCond*$_1$*-atomics*:
$[\![\ interfree\text{-}aux\ (c,\ q,\ Some\ a1);\ interfree\text{-}aux\ (c,\ q,\ Some\ a2)\ ]\!] \Longrightarrow$
$interfree\text{-}aux\ (c,\ q,\ Some\ (AnnCond_1\ r\ b\ a1\ a2))$
lemma *AnnCond*$_2$*-atomics*:
$interfree\text{-}aux\ (c,\ q,\ Some\ a) \Longrightarrow interfree\text{-}aux\ (c,\ q,\ Some\ (AnnCond_2\ r\ b\ a))$
lemma *AnnWhile-atomics*:
$interfree\text{-}aux\ (c,\ q,\ Some\ a) \Longrightarrow interfree\text{-}aux\ (c,\ q,\ Some\ (AnnWhile\ r\ b\ i\ a))$
lemma *Annatom-atomics*:
$\Vdash (q \cap r)\ a\ q \Longrightarrow interfree\text{-}aux\ (None,\ q,\ Some\ (AnnAwait\ r\ \{x.\ True\}\ a))$
lemma *AnnAwait-atomics*:
$\Vdash (q \cap (r \cap b))\ a\ q \Longrightarrow interfree\text{-}aux\ (None,\ q,\ Some\ (AnnAwait\ r\ b\ a))$

By repeatedly applying these rules backwards, only subgoals that state the derivability of a Hoare-triple remain. Then, the verification conditions are automatically generated with the vcg-tactic for atomic programs *HoareRuleTac* since the interference freedom tests involve only atomic actions.

The full tactic for the generation of the verification conditions of a parallel program is a combination of the tactics and rules above. Because of the interdependencies between them they must be defined together connected by the keyword *and*[2]:

ML {*

```
fun WlpTac i = rtac (thm SeqRule) i THEN HoareRuleTac false (i+1)
and HoareRuleTac precond i =
      WlpTac i THEN HoareRuleTac precond i
    ORELSE
     FIRST[rtac (thm SkipRule) i,
           rtac (thm BasicRule) i,
           EVERY[rtac (thm ParallelConseqRule) i,
                 ParallelTac (i+1)],
           EVERY[rtac (thm CondRule) i,
                 HoareRuleTac false (i+2),
```

[2]This tactic is a slight simplification of the real one which can be found in the source theories.

```
                    HoareRuleTac false (i+1)],
              EVERY[rtac (thm WhileRule) i,
                    HoareRuleTac true (i+1)]]
        THEN (if precond then (K all-tac i) else rtac (thm subset-refl) i)

and AnnWlpTac i = rtac (thm AnnSeq) i
                  THEN AnnHoareRuleTac (i+1)
and AnnHoareRuleTac i =
      AnnWlpTac i THEN AnnHoareRuleTac i
    ORELSE
      FIRST[rtac (thm AnnSkipRule) i,
              EVERY[rtac (thm AnnAtomRule) i,
                    HoareRuleTac true (i+1)],
              rtac (thm AnnWaitRule) i,
              rtac (thm AnnBasic) i,
              EVERY[rtac (thm AnnCond$_1$) i,
                    AnnHoareRuleTac (i+3),
                    AnnHoareRuleTac (i+1)],
              EVERY[rtac (thm AnnCond$_2$) i,
                    AnnHoareRuleTac (i+1)],
              EVERY[rtac (thm AnnWhile) i,
                    AnnHoareRuleTac (i+2)],
              EVERY[rtac (thm AnnAwait) i,
                    HoareRuleTac true (i+1)]]

and ParallelTac i = EVERY[rtac (thm ParallelRule) i,
                          interfree-Tac (i+1),
                          MapAnn-Tac i]

and MapAnn-Tac i =
    FIRST[rtac (thm MapAnnEmpty) i,
            EVERY[rtac (thm MapAnnList) i,
                  MapAnn-Tac (i+1),
                  AnnHoareRuleTac i],
            EVERY[rtac (thm MapAnnMap) i,
                  rtac (thm allI) i, rtac (thm impI) i,
                  AnnHoareRuleTac i]]

and interfree-swap-Tac i =
    FIRST[rtac (thm interfree-swap-Empty) i,
          EVERY[rtac (thm interfree-swap-List) i,
                interfree-swap-Tac (i+2),
                interfree-aux-Tac (i+1),
                interfree-aux-Tac i ],
          EVERY[rtac (thm interfree-swap-Map) i,
                rtac (thm allI) i, rtac (thm impI) i, rtac (thm conjI) i,
                interfree-aux-Tac (i+1),
                interfree-aux-Tac i]]

and interfree-Tac i =
    FIRST[rtac (thm interfree-Empty) i,
          EVERY[rtac (thm interfree-List) i,
                interfree-Tac (i+1),
                interfree-swap-Tac i],
          EVERY[rtac (thm interfree-Map) i,
                 rtac (thm allI) i, rtac (thm allI) i, rtac (thm impI) i,
                 interfree-aux-Tac i ]]

and interfree-aux-Tac i =
    FIRST[rtac (thm interfree-aux-rule1) i,
          dest-assertions-Tac i]

and dest-assertions-Tac i =
    FIRST[EVERY[rtac (thm AnnBasic-assertions) i,
                dest-atomics-Tac (i+1),
                dest-atomics-Tac i],
          EVERY[rtac (thm AnnSeq-assertions) i,
                dest-assertions-Tac (i+1),
                dest-assertions-Tac i],
          EVERY[rtac (thm AnnCond_1-assertions) i,
                dest-assertions-Tac (i+2),
                dest-assertions-Tac (i+1),
                dest-atomics-Tac i],
          EVERY[rtac (thm AnnCond_2-assertions) i,
                dest-assertions-Tac (i+1),
                dest-atomics-Tac i],
          EVERY[rtac (thm AnnWhile-assertions) i,
                dest-assertions-Tac (i+2),
                dest-atomics-Tac (i+1),
                dest-atomics-Tac i],
            EVERY[rtac (thm AnnAwait-assertions) i,
                  dest-atomics-Tac (i+1),
                  dest-atomics-Tac i],
            dest-atomics-Tac i]

and dest-atomics-Tac i =
    FIRST[EVERY[rtac (thm AnnBasic-atomics) i,
                 HoareRuleTac true i],
            EVERY[rtac (thm AnnSeq-atomics) i,
                  dest-atomics-Tac (i+1),
                  dest-atomics-Tac i],
            EVERY[rtac (thm AnnCond_1-atomics) i,
                  dest-atomics-Tac (i+1),
                  dest-atomics-Tac i],
            EVERY[rtac (thm AnnCond_2-atomics) i,
                  dest-atomics-Tac i],
            EVERY[rtac (thm AnnWhile-atomics) i,
                  dest-atomics-Tac i],
            EVERY[rtac (thm Annatom-atomics) i,
                  HoareRuleTac true i],
            EVERY[rtac (thm AnnAwait-atomics) i,
                  HoareRuleTac true i]]
*}
```

Note that subgoals are always treated counting downwards, to avoid problems when subgoals are added or deleted. The final tactic is given the name *oghoare*:

```
ML {*
fun oghoare-tac i thm = SUBGOAL (fn (term, -) =>
  (HoareRuleTac true i)) i thm
*}
```

Notice that the tactic for parallel programs *oghoare-tac* is initially invoked with the value *true* for the parameter *precond*.

Parts of the tactic can be also individually used to generate the verification conditions for annotated sequential programs and to generate verification conditions out of interference freedom tests like in §3.3.3:

```
ML {*
fun annhoare-tac i thm = SUBGOAL (fn (term, -) =>
  (AnnHoareRuleTac i)) i thm

fun interfree-aux-tac i thm = SUBGOAL (fn (term, -) =>
  (interfree-aux-Tac i)) i thm
*}
```

The so defined ML tactics are then "exported" to be used in Isabelle proofs.

```
method-setup oghoare = {*
  Method.no-args (Method.SIMPLE-METHOD′ HEADGOAL (oghoare-tac)) *}
  verification condition generator for the oghoare logic


method-setup annhoare = {*
  Method.no-args
    (Method.SIMPLE-METHOD′ HEADGOAL (annhoare-tac)) *}
  verification condition generator for the ann-hoare logic

method-setup interfree-aux = {*
  Method.no-args
    (Method.SIMPLE-METHOD′ HEADGOAL (interfree-aux-tac)) *}
  verification condition generator for interference freedom tests
```

The three tactics for generating verification conditions, *oghoare* for parallel programs, *annhoare* for annotated sequential programs and *interfree-aux* for parts of interference freedom tests, are the only tactics used to verify the examples presented throughout this work. They are invoked simply with **apply**, i.e. **apply** *oghoare*, etc.

Appendix B

Formal Declaration of Concrete Syntax

This section presents the specification of syntax and translations needed to obtain a user-friendly external representation for programs and assertions. As explained in §2.7.1 we use the quote/antiquote technique.

syntax
  -*quote* :: $\beta \Rightarrow (\alpha \Rightarrow \beta)$ (≪-≫)
  -*antiquote* :: $(\alpha \Rightarrow \beta) \Rightarrow \beta$ (´-)

The syntax and translations declared here correspond to the programming language used for the Owicki-Gries formalization. The counterpart for the language of the rely-guarantee formalization is very similar and thus not shown here.

We start with the syntax for component programs. Mixfix notation for commands can be defined by declaring new syntax constants.

syntax
  -*Assign* :: *idt* $\Rightarrow \beta \Rightarrow \alpha$ *com* (´- := -)
  -*Seq* :: α *com* $\Rightarrow \alpha$ *com* $\Rightarrow \alpha$ *com* (-,,/ -)
  -*Cond* :: α *bexp* $\Rightarrow \alpha$ *com* $\Rightarrow \alpha$ *com* $\Rightarrow \alpha$ *com* (**if** - **then** - **else** - **fi**)
  -*While-inv* :: α *bexp* $\Rightarrow \alpha$ *assn* $\Rightarrow \alpha$ *com* $\Rightarrow \alpha$ *com* (**while** - **inv** - **do** - **od**)

  -*AnnAssign* :: α *assn* $\Rightarrow$ *idt* $\Rightarrow \beta \Rightarrow \alpha$ *com* (- ´- := -)
  -*AnnSeq* :: α *ann-com* $\Rightarrow \alpha$ *ann-com* $\Rightarrow \alpha$ *ann-com* (-;;/ -)
  -*AnnCond*$_1$ :: α *assn* $\Rightarrow \alpha$ *bexp* $\Rightarrow \alpha$ *ann-com* $\Rightarrow \alpha$ *ann-com*

$\qquad\qquad \Rightarrow \alpha\ \mathit{ann\text{-}com}\ \ (\text{-}\ \mathbf{if}\ \text{-}\ \mathbf{then}\ \text{-}\ \mathbf{else}\ \text{-}\ \mathbf{fi})$
$\text{-}\mathit{AnnCond}_2 :: \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{bexp}\ \Rightarrow \alpha\ \mathit{ann\text{-}com} \Rightarrow \alpha\ \mathit{ann\text{-}com}\ (\text{-}\ \mathbf{if}\ \text{-}\ \mathbf{then}\ \text{-}\ \mathbf{fi})$
$\text{-}\mathit{AnnWhile} :: \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{bexp}\ \Rightarrow \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{ann\text{-}com} \Rightarrow \alpha\ \mathit{ann\text{-}com}$
$\qquad\qquad (\text{-}\ \mathbf{while}\ \text{-}\ \mathbf{inv}\ \text{-}\ \mathbf{do}\ \text{-}\ \mathbf{od})$
$\text{-}\mathit{AnnAwait} :: \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{bexp}\ \Rightarrow \alpha\ \mathit{com} \Rightarrow \alpha\ \mathit{ann\text{-}com}$
$\qquad\qquad (\text{-}\ \mathbf{await}\ \text{-}\ \mathbf{then}\ \text{-}\ \mathbf{end})$

We also introduce external syntax for commands which are simply abbreviations of the existing ones.

syntax
$\text{-}\mathit{Skip} :: \alpha\ \mathit{com}\ \ (\mathbf{skip})$
$\text{-}\mathit{Cond}_2 :: \alpha\ \mathit{bexp} \Rightarrow \alpha\ \mathit{com} \Rightarrow \alpha\ \mathit{com}\ \ \ (\mathbf{if}\ \text{-}\ \mathbf{then}\ \text{-}\ \mathbf{fi})$

$\text{-}\mathit{AnnSkip} :: \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{ann\text{-}com}\ \ (\text{-}\ \mathbf{skip})$
$\text{-}\mathit{AnnAtom} :: \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{com} \Rightarrow \alpha\ \mathit{ann\text{-}com}\ \ \ (\text{-}\ \langle\text{-}\rangle)$
$\text{-}\mathit{AnnWait} :: \alpha\ \mathit{assn} \Rightarrow \alpha\ \mathit{bexp} \Rightarrow \alpha\ \mathit{ann\text{-}com}\ \ \ (\text{-}\ \mathbf{wait}\ \text{-}\ \mathbf{end})$

The external syntax for assertions is:

syntax
$\text{-}\mathit{Assert} :: \alpha \Rightarrow \alpha\ \mathit{set}\ \ \ (\{\!|\text{-}|\!\})$

Part of the corresponding translations of these concrete syntax constants into internal (abstract) syntax can be done simply by directed rewriting equations:

translations
$\{\!|b|\!\} \rightharpoonup \mathit{Collect} \ll b \gg$

$\acute{}x := a \rightharpoonup \mathit{Basic} \ll \acute{}(\text{-}\mathit{update\text{-}name}\ x\ a) \gg$
$c_1,,\ c_2 \rightleftharpoons \mathit{Seq}\ c_1\ c_2$
$\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \rightharpoonup \mathit{Cond}\ \{\!|b|\!\}\ c_1\ c_2$
$\mathbf{while}\ b\ \mathbf{inv}\ i\ \mathbf{do}\ c\ \mathbf{od} \rightharpoonup \mathit{While}\ \{\!|b|\!\}\ i\ c$

$r\ \acute{}x := a \rightharpoonup \mathit{AnnBasic}\ r \ll \acute{}(\text{-}\mathit{update\text{-}name}\ x\ a) \gg$
$c_1;;\ c_2 \rightleftharpoons \mathit{AnnSeq}\ c_1\ c_2$
$r\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \rightharpoonup \mathit{AnnCond}_1\ r\ \{\!|b|\!\}\ c_1\ c_2$
$r\ \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{fi} \rightharpoonup \mathit{AnnCond}_2\ r\ \{\!|b|\!\}\ c$
$r\ \mathbf{while}\ b\ \mathbf{inv}\ i\ \mathbf{do}\ c\ \mathbf{od} \rightharpoonup \mathit{AnnWhile}\ r\ \{\!|b|\!\}\ i\ c$
$r\ \mathbf{await}\ b\ \mathbf{then}\ c\ \mathbf{end} \rightharpoonup \mathit{AnnAwait}\ r\ \{\!|b|\!\}\ c$

skip $\rightleftharpoons$ *Basic id*
if b **then** c **fi** $\rightleftharpoons$ **if** b **then** c **else skip fi**

r **skip** $\rightleftharpoons$ *AnnBasic* r *id*
r $\langle c \rangle$ $\rightleftharpoons$ r **await** *True* **then** c **end**
r **wait** b **end** $\rightleftharpoons$ r **await** b **then skip end**

These equations translate in the direction of the arrow. Expressions on the left-hand side are the input syntax, and those on the right-hand side are the internal representation. The one-to-one translations, denoted by $\rightleftharpoons$ are automatically performed by Isabelle's parser and printer. This is the case for the sequential composition and the command's abbreviations. For the rest of the commands the translation from internal to external representation is a little more complicated because boolean conditions and program variables have to be translated according to the quote/antiquote technique. When the translations are too complicated as to be expressed by an equation, Isabelle allows so-called parse and print translation functions to be programmed in ML.

For parallel programs there is also some concrete syntax of the form

cobegin c_1 $\{\!|q1|\!\}$ $\|$ $\ldots$ $\|c_n\{\!|q_n|\!\}$ **coend**

defined formally as follows:

nonterminals
prgs
syntax
-PAR :: *prgs* $\Rightarrow \alpha$ (**cobegin** - **coend**)
-prg :: $[\alpha, \alpha] \Rightarrow$ *prgs* (- -)
-prgs :: $[\alpha, \alpha,$ *prgs*$] \Rightarrow$ *prgs* (- - $\|$ -)

The following one-direction translations translate the external representation into a *Parallel* command with argument the corresponding list of component programs:

translations
-prg a c $\rightharpoonup$ $[(Some\ a,\ c)]$
-prgs a c ps $\rightharpoonup$ $(Some\ a,\ c)$ # *ps*
-PAR ps $\rightharpoonup$ *Parallel ps*

The definition of syntax and the corresponding translation for program schemas are:

syntax
  -prg-scheme :: $[\alpha, \alpha, \alpha, \alpha, \alpha] \Rightarrow prgs$ (**scheme** [- $\leq$ - $<$ -] - -)
translations
  -prg-scheme $j\ i\ k\ c\ q \rightleftharpoons (map\ (\lambda i.\ (Some\ c,\ q))\ [j..k(])$

Notice that the internal representation is just a list of parallel programs. Thus, it has to be enclosed in a **cobegin**− **coend** environment, where it can be further composed with other parameterized or concrete parallel programs.

With these equations, the translation from external into internal representation is almost finished, only one constant is still not understandable for Isabelle, namely *-quote*. To translate quotations into internal syntax we use a *parse-translation* function which uses the basic quote/antiquote translations predefined in the theory Isabelle/Pure (see `Syntax.quote_tr` and `Syntax.quote_tr'`).

parse-translation {∗
```
  let
    fun quote-tr [t] = Syntax.quote-tr -antiquote t
      | quote-tr ts = raise TERM (quote-tr, ts);
  in [(-quote, quote-tr)] end
∗}
```

This last step completes the translation from external into internal syntax.

For the recovery of the external syntax from the internal representation, a similar ML program called *print-translation* is defined. As usual in Isabelle syntax translations, the part for printing is more complicated. It is only recommended for Isabelle experts and thus it is not shown here. However, we mention that the full ML *print-translation* function consists only of 50 lines of code.

As a comparative remark, it is worth mentioning that the ML program implementing the parse and print translations needed for a method to represent the state via abstraction over tuples of program variables due to [von Wright *et al.*, 1993] (see 2.7.1) consisted of 550 lines of code.

Bibliography

[Abdulla and Jonsson, 1998] P. A. Abdulla and B. Jonsson. Verifying networks of timed processes. In B. Steffen, editor, 7^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lect. Notes in Comp. Sci.*, pages 298–312, 1998.

[Alur and Dill, 1994] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994.

[Andersen *et al.*, 1994] F. Andersen, K. Petersen, and J. Pettersson. Program verification using HOL-UNITY. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lect. Notes in Comp. Sci.*, pages 1–15. Springer-Verlag, 1994.

[Apt and Kozen, 1986] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[Apt and Olderog, 1991] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[Apt *et al.*, 1980] K. Apt, N. Francez, and W. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[Apt, 1981a] K. R. Apt. Recursive assertions and parallel programs. *Acta Informatica*, 15:219–232, 1981.

[Apt, 1981b] K. R. Apt. Ten years of hoare logic: A survey – part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.

[Basten and Hooman, 1999] T. Basten and J. Hooman. Process algebra in PVS. In *Proceedings TACAS '99*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 270–284. Springer-Verlag, 1999.

[Ben-Ari, 1984] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans. Programming Languages and Systems*, 6:333–344, 1984.

[Best, 1996] E. Best. *Semantics of Sequential and Parallel Programs.* International Series in Computer Science. Prentice Hall, 1996.

[Bruns, 1997] G. Bruns. *Distributed Systems Analysis with CCS.* Prentice-Hall, 1997.

[Camillieri, 1990] A. Camillieri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16:993–1004, 1990.

[Chandy and Misra, 1984] K. Chandy and J. Misra. The drinking-philosophers problem. In *ACM Trans. Programming Languages and Systems*, volume 6, pages 632–646, 1984.

[Chetali and Heyd, 1997] B. Chetali and B. Heyd. Formal verification of concurrent programs in LP and COQ: A comparative analysis. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 69–85. Springer-Verlag, 1997.

[Clarke *et al.*, 1995] E. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In Lee and Smolka, editors, 6^{th} *Int. Conf. on Concurrency Theory (CONCUR '95)*, volume 962 of *Lect. Notes in Comp. Sci.*, pages 395–407, 1995.

[Das *et al.*, 1999] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 160–171. Springer-Verlag, 1999.

[de Boer *et al.*, 1997] F. de Boer, U. Hannemann, and W.-P. de Roever. Hoare-style compositional proof systems for reactive shared variable concurrency. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1997.

[de Roever *et al.*, 1998] W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference, Proceedings of the International Symposium COMPOS '97*, volume 1536 of *Lect. Notes in Comp. Sci.*, Malente, Germany, 7–12 September 1997 1998. Springer-Verlag.

[de Roever *et al.*, 2000] W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *State-Based Proof Theory of Concurrency: from Noncompositional to Compositional Methods.* Tracts in Theoretical Computer Science. Cambridge University Press, 2000. To appear.

[Dijkstra *et al.*, 1978] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[Dijkstra, 1968] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112, London, 1968. Academic Press.

[Dijkstra, 1976] E. W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[Dutertre and Schneider, 1997] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 121–136. Springer-Verlag, 1997.

[Engberg *et al.*, 1993] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. Probst, editors, *Computer-Aided Verification (CAV '92)*, volume 663 of *Lect. Notes in Comp. Sci.*, pages 44–55. Springer-Verlag, 1993.

[Esparza, 1995] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. In *Fundamentals of Computation Theory*, volume 965 of *Lect. Notes in Comp. Sci.*, pages 221–232, 1995.

[Feijen and van Gasteren, 1999] W. Feijen and A. van Gasteren. On a method of multiprogramming. In *Monographs in Computer Science.* Springer-Verlag, 1999.

[Filliatre, 1999] J.-C. Filliatre. *Preuve de programmes impératifs en théorie des types.* PhD thesis, Université Paris-Sud, 1999.

[Francez and Rodeh, 1980] N. Francez and M. Rodeh. Achieving distributed termination without freezing. Technical report (TR 72), IBM Israel Scientific Center, 1980.

[Galm, 1995] N. Galm. Verifikation von IMP-Programmen mit Hilfe der Hoare-Logik mit dem Theorembeweiser Isabelle. Ausarbeitung zum Fortgeschrittenen-Praktikum, 1995.

[German and Sistla, 1992] S. German and A. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[Goldschlag, 1990] D. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16:1005–1022, 1990.

[Gordon and Melham, 1993] M. Gordon and T. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Gordon, 1979] M. Gordon. *The Denotational Description of Programming Languages, An Introduction*. Springer-Verlag, New York, 1979.

[Gordon, 1989] M. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[Gries, 1997] D. Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, 1997.

[Harrison, 1998] J. Harrison. Formalizing Dijkstra. In *Theorem Proving in Higher Order Logics (TPHOLs '98)*, volume 1497 of *Lect. Notes in Comp. Sci.*, pages 171–188. Springer-Verlag, 1998.

[Havelund and Shankar, 1997] K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. *Formal Aspects of Computing*, 3:1–28, 1997.

[Havelund, 1996] K. Havelund. Mechanical verification of a garbage collector. In *FMPPTA*, 1996. Available at http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund/.

[Hennessy and Plotkin, 1979] M. Hennessy and G. Plotkin. Full abstraction for a simple programming language. In *Mathematical Foundations of Computer Science*, volume 74 of *Lect. Notes in Comp. Sci.*, pages 108–120. Springer-Verlag, 1979.

[Henzinger, 1995] T. Henzinger. Hybrid automata with finite bisimulations. In *ICALP '95*, 1995.

[Heyd and Crégut, 1996] B. Heyd and P. Crégut. A modular coding of Unity in Coq. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 251–266. Springer-Verlag, 1996.

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

[Homeier and Martin, 1996] P. V. Homeier and D. F. Martin. Mechanical verification of mutually recursive procedures. In M. McRobbie and J. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lect. Notes in Comp. Sci.*, pages 201–215. Springer-Verlag, 1996.

[Hooman *et al.*, 2000] J. Hooman, W.-P. de Roever, P. Pandya, H. Schepers, Q. Xu, and P. Zhou. *Compositional Theory of Concurrency.* Cambridge University Press, 2000. To appear.

[Hooman, 1995] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In P. S. Thiagarajan, editor, 15^{th} *Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lect. Notes in Comp. Sci.*, pages 96–110, Bangalore, India, December 1995. Springer-Verlag.

[Hooman, 1998] J. Hooman. Developing proof rules for distributed real-time systems with PVS. In *Proc. Workshop on Tool Support for System Development and Verification*, volume 1 of *BISS Monographs*, pages 120–139. Shaker Verlag, 1998.

[Jackson, 1998] P. Jackson. Verifying a garbage collection algorithm. In J. Grundy and M. Newey, editors, *11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '98)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Available at `www.dcs.ed.ac.uk/home/pbj`.

[Jones, 1981] C. B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University Computing Laboratory, 1981.

[Jones, 1983] C. B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jonker, 1992] J. E. Jonker. On-the-fly garbage collection for several mutator. *Distributed Computing*, 5:187–199, 1992.

[Jonsson and Parrow, 1993] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite state programs. *Information and Computation*, 107(2):272–302, 1993.

[Kalvala, 1995] S. Kalvala. A formulation of TLA in isabelle. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 214–228. Springer-Verlag, 1995.

[Kleymann, 1998] T. Kleymann. *Hoare logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, Department of Computer Science, Univ. of Edinburgh, 1998.

[Lamport, 1977] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Långbacka and von Wright, 1997] T. Långbacka and J. von Wright. Refining reactive systems in HOL using action systems. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 183–197. Springer-Verlag, 1997.

[Levin and Gries, 1981] G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[Misra and Chandy, 1981] J. Misra and M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engeneering*, 7(7):417–426, 1981.

[Müller and Nipkow, 1997] O. Müller and T. Nipkow. Traces of I/O automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 580–594. Springer-Verlag, 1997.

[Müller, 1998] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, TU München, September 1998.

[Naraschewski and Wenzel, 1998] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.

[Nesi, 1994] M. Nesi. Value-passing CCS in HOL. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lect. Notes in Comp. Sci.*, pages 352–365. Springer-Verlag, 1994.

[Nipkow and Paulson, 2001] T. Nipkow and L. C. Paulson. Isabelle/HOL – The Tutorial. Available at http://isabelle.in.tum.de/doc/tutorial.pdf, 2001.

[Nipkow and Prensa Nieto, 1999] T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE '99)*, volume 1577 of *Lect. Notes in Comp. Sci.*, pages 188–203. Springer-Verlag, 1999.

[Nipkow, 1996] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.

[Nipkow, 1998] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

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

[Owicki and Gries, 1976b] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, 1976.

[Owicki, 1975] S. Owicki. *Axiomatic proof techniques for parallel programs.* PhD thesis, Computer Science Dept., Cornell University, 1975.

[Owre *et al.*, 1995] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Owre *et al.*, 1996] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lect. Notes in Comp. Sci.*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Paulson, 1994] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. Isabelle home page: `http://isabelle.in.tum.de/`.

[Paulson, 2000] L. C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, 2000.

[Paulson, 2001] L. C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 2001. in Press.

[Peterson, 1981] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):223–252, 1981.

[Pixley, 1988] C. Pixley. An incremental garbage collection algorithm for multimutator systems. *Distributed Computing*, 3:41–50, 1988.

[Plotkin, 1981] G. D. Plotkin. A structural approach to operational semantics. Technical report DAIMI-FN 19, Department of Computer Science, Aarhus University, 1981.

[Prensa Nieto and Esparza, 2000] L. Prensa Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS '00)*, volume 1893 of *Lect. Notes in Comp. Sci.*, pages 619–628. Springer-Verlag, 2000.

[Prensa Nieto, 2001] L. Prensa Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In *Formal Methods for Parallel Programming: Theory and Applications (FMPPTA '01)*, 2001.

[Rogers, 1987] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, 1987.

[Rushby, 2000] J. Rushby. Theorem proving for verification. In F. Cassez, editor, *Modelling and Verification of Parallel Processes: MoVEP 2k*, Nantes, France, June 2000. Tutorial presented at MoVEP: revised version to be published by Springer LNCS.

[Russinoff, 1994] D. M. Russinoff. A mechanically verified garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[Schoenfield, 1967] J. R. Schoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[Scott and Strachey, 1971] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical report PRG–6, Programming Research Group, University of Oxford, 1971.

[Shankar, 1996] N. Shankar. PVS: Combining specification, proof checking, and model checking. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lect. Notes in Comp. Sci.*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.

[Shankar, 1998] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy, editor, *Mathematical Methods in Program Development*. Springer-Verlag, 1998.

[Søgaard-Andersen *et al.*, 1993] J. Søgaard-Andersen, S. Garland, J. Guttag, N. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In 4^{th} *Conference on Computer-Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 305–319. Springer-Verlag, 1993.

[Soundararajan, 1984] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31:13–29, 1984.

[Stirling, 1988] C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

[Stølen, 1990] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Computer Science Department, Manchester University, 1990.

[Stølen, 1991] K. Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91*, volume 527 of *Lect. Notes in Comp. Sci.*, pages 510–525. Springer-Verlag, 1991.

[Tej and Wolff, 1997] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lect. Notes in Comp. Sci.*, pages 318–337. Springer-Verlag, 1997.

[van de Snepscheut, 1987] J. L. A. van de Snepscheut. "Algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24:211–216, 1987.

[von Oheimb, 2001] D. von Oheimb. *Analyzing Java in Isabelle/HOL – Formalization, Type Safety and Hoare Logic*. PhD thesis, TU München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.

[von Wright and Långbacka, 1993] J. von Wright and T. Långbacka. Using a theorem prover for reasoning about concurrent algorithms. In G. v. Bochmann and D. Probst, editors, *Computer-Aided Verification (CAV '92)*, volume 663 of *Lect. Notes in Comp. Sci.*, pages 56–68. Springer-Verlag, 1993.

[von Wright *et al.*, 1993] J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.

[Wenzel, 2001a] M. Wenzel. *Isabelle/Isar – a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, 2001. `http://www4.in.tum.de/~wenzelm/diss/`.

[Wenzel, 2001b] M. Wenzel. Miscellaneous Isabelle/Isar examples for higher order logic. Isabelle/Isar proof document, TU München, February 2001.

[Wilson, 1992] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, Lect. Notes in Comp. Sci., 1992. Available at `www.cs.ukc.ac.uk/people/staff/rej/gc.html`.

[Winskel, 1993] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[Xu *et al.*, 1995] Q. Xu, W.-P. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. Technical report, Christian-Albrechts-Universität Kiel, 1995.

[Xu *et al.*, 1997] Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

Index