Owicki/Gries in Isabelle/HOL

Tobias Nipkow and Leonor Prensa Nieto

Technische Universität München
Institut für Informatik, 80290 München, Germany
`http://www.in.tum.de/~{nipkow,prensani}`

Abstract. We present a formalization of the Gries/Owicki method for correctness proofs of concurrent imperative programs with shared variables in the theorem prover Isabelle/HOL. Syntax, semantics and proof rules are defined in higher-order logic. The correctness of the proof rules w.r.t. the semantics is proved. The verification of some typical example programs like producer/consumer is presented.

1 Introduction

This paper presents the first formalization in a theorem prover (namely Isabelle/HOL) of the well-known Owicki/Gries method [22] of correctness proofs for shared-variable concurrency. The programming language is a simple `WHILE`-language with concurrent execution of commands and synchronization via an `AWAIT` command. We define the operational semantics and the Owicki/Gries proof system and prove the soundness of the latter w.r.t. the former. Our soundness proof is interesting because it does not explicitly mention program locations, as is customary in the literature. Based on the proof rules we develop a verification condition generator (as an Isabelle tactic). Finally we present some typical examples, including the verification of a schematic program where the number of parallel components is a parameter. In particular this example shows that our embedding is more than a glorified verification condition generator: the availability of a full blown theorem prover means we can tackle problems outside the range of automatic methods like model checking.

One can also consider this paper a continuation of the first author's work on formalizing textbooks on programming language semantics [19, 21] because we have closely followed the description of Owicki/Gries in [2]. However, this is only a minor aspect and we are more interested in the practical application of our setup rather than meta-theoretical properties. Therefore we have proved soundness but not completeness of the proof system. Instead, we have concentrated on automating verification condition generation and on matters of surface syntax, which are frequently ignored in similar endeavours.

We are well aware of the non-compositionality of the Owicki/Gries method, and consider this only as a first step towards compositional methods. However, we wanted to start with the most fundamental approach rather than pick one of several competing compositional systems. We believe that a full understanding of the technicalities (from the theorem-proving point of view) involved in formalizing Owicki/Gries is advantageous when tackling compositional proof systems.

2 Related work

The idea of defining the syntax, semantics and proof system of an imperative programming language in a theorem prover goes back at least to Gordon [9], who considered the Hoare-logic of a simple `WHILE`-language. Stimulated by Gordon's paper, a fair number of concurrency paradigms have been formalized in theorem provers: UNITY is the most frequently chosen framework and has been formalized in the Boyer-Moore prover [8], HOL [1], Coq [11] and LP [4]. A related framework, *action systems*, has also been formalized in HOL [16]. CSP has been treated in HOL [3], Isabelle/HOL [26] and PVS [6]. CCS has been formalized in HOL [18]. TLA has been formalized in HOL [28], LP [7] and Isabelle/HOL [15]. Input/Output Automata have been formalized in Isabelle/HOL [17] and in LP [24]. However, it appears that there has been no work on embedding Hoare-logics for shared-variable parallelism in a 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. Further important steps were the compositional system for shared-variable parallelism put forward by Jones [13, 14] (the rely/guarantee method) and the complete version of this system designed by Stølen [25]. For more recent work on the subject see [5].

3 Isabelle/HOL

Isabelle [23, 12] 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 [10]. From now on HOL refers to Isabelle/HOL. For a tutorial introduction see [20]. We do not assume that the reader is already familiar with HOL and summarize the relevant notation below.

The type system is similar to ML, except that function types are denoted by $\Rightarrow$. List notation is also similar to ML (e.g. `@` is 'append') except that the 'cons' operation is denoted by `#` instead of `::`. The ith component of list xs is written $xs\,!\,i$, and $xs\,[i:=x]$ denotes xs with the ith component replaced by x.

Set comprehension syntax is $\{e.\ P\}$.

The notation $[\![A_1\,;\ldots;A_n]\!] \Longrightarrow A$ represents an implication with assumptions $A_1,\ldots,A_n$ and conclusion A.

To distinguish variables from constants, the latter are shown in $\mathsf{sans\text{-}serif}$.

4 The Programming Language

Our programming language is a `WHILE`-language augmented with shared-variable parallelism ($\|$) and synchronization (`AWAIT`). We follow [2] in stratifying the language: parallelism must not be nested, i.e. each c_i in $c_1\|\ldots\|c_n$ must be a sequential command. In HOL, there are two ways to encode this stratification: define the type of all programs and a predicate that excludes those programs with nested parallelism, or define the syntax in two layers. We have chosen the latter alternative because it eliminates the well-formedness predicate, thus simplifying statements and proofs about the language. The fact that the language specification becomes longer because a number of constructs appear in both layers is uncritical: although proofs about those constructs have to be duplicated, this duplication is quite mechanical.

4.1 Syntax versus Semantics

How much of the syntax of the programming language do we want to represent in HOL? The choice is between a **deep embedding**, where the (abstract) syntax is represented via an inductive datatype, and a **shallow embedding**, where a term in the language is merely an abbreviation of its semantics. Deep embeddings are required for meta-theoretic reasoning (usually by induction over the syntax), whereas shallow embeddings simplify reasoning about individual programs (because the semantics is directly given) but may rule out certain meta-theoretic arguments. We have chosen a combination of both styles that has become quite established: represent as much as possible by a shallow embedding while still being able to conduct the meta-theoretic proofs you are interested in. Concretely this means that we use a deep embedding for programs themselves, whereas assertions, expressions and even assignments are represented semantically, i.e. as functions on *states*. For the time being we do not say more about the nature of states and merely define the parameterized type abbreviations

```
(α)assn = (α)set
(α)bexp = (α)set
```

representing both assertions and boolean expressions as sets (of states). The precise representation of states is discussed in §4.3 below. We also define the constant TRUE to be the universal set.

4.2 Component Programs

The core language is a standard sequential `WHILE`-language augmented with a synchronization construct (`AWAIT`). We depart from the usual presentation of the language by including assertions directly in the syntax: every construct, apart from sequential composition, is annotated with a precondition, and `WHILE` is also annotated with an invariant. We emphasize that these assertions are merely annotations and do not change the semantics of the language. Their sole purpose is to record proof outlines which can later be checked for interference freedom. Since we aim at a formalism for designing correct programs, it is only reasonable to include the necessary assertions right from the start. Moreover, it turns out that for proof-theoretic reasons it is very helpful to define the semantics of the language directly on annotated commands.

As discussed above, annotated commands are defined as a datatype with one constructor for each syntactic construct:

```
α ann_com = AnnBasic (α assn) (α ⇒ α)
          | AnnSeq (α ann_com) (α ann_com)
               ("_ ;; _")
          | AnnCond (α assn) (α bexp) (α ann_com) (α ann_com)
               ("_ IF _ THEN _ ELSE _ FI")
          | AnnWhile (α assn) (α bexp) (α assn) (α ann_com)
               ("_ WHILE _ INV _ DO _ OD")
          | AnnAwait (α assn) (α bexp) (α atom_com)
              ("_ AWAIT _ THEN _ END")
```

The optional ("...") annotations describe the obvious concrete syntax. We discuss the different constructs in turn, ignoring their preconditions.

AnnBasic represents a basic atomic state transformation, for example an assignment, a multiple assignment, or even any non-constructive specification. It turns out that it is quite immaterial what the basic commands are. The concrete syntax of assignments is explained in §4.3 below.

AnnSeq is the sequential composition of commands. We use `;;` in the concrete syntax to avoid clashes with the predefined `;` in Isabelle.

AnnCond is the conditional.

AnnWhile is the loop, annotated with an invariant.

AnnAwait is the synchronization construct. Following [2], its body must be loop-free. Type `atom_com` of loop-free commands is described in §4.4 below.

The precondition of each annotated command is extracted by the function `pre:: α ann_com ⇒ α assn`. Its definition is obvious and we merely show one clause: `pre(c1;;c2) = pre(c1)`.

The semantics of annotated commands is inductively defined by a set of transition rules between configurations, where a configuration is a pair of a program fragment and a state. A program fragment is either an annotated command, or, if execution has come to an end, the empty program. Adjoining a new element to a type is naturally modeled by the standard datatype

```
(α)option = None | Some α
```

In our case, None represents the empty program. The transition rules are shown in Fig. 1, where $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow^1$, and $\rightarrow_a^*$ is the atomic execution of loop-free commands described in §4.4 below.

4.3 The State and Concrete Syntax

The state during program execution is often modeled as a mapping from variables to values. This is fine in systems with dependent function spaces, but in the case of HOL's simple function spaces it means that all variables must be of the same type. There are two dual ways out of this: a mapping from variables to the disjoint union of all types used in the program, or a tuple of values, one for each variable occurring in the program. Note that in both cases, the state of the

```
(Some(AnnBasic R f), s) →¹ (None, f s)

(Some(c₀), s)       →¹ (None, t) ⟹
(Some(c₀;;c₁), s) →¹ (Some(c₁), t)

(Some(c₀), s)       →¹ (Some(c₂), t) ⟹
(Some(c₀;;c₁), s) →¹ (Some(c₂;;c₁), t)

s ∈ b  ⟹  (Some(R IF b THEN c₁ ELSE c₂ FI), s) →¹ (Some(c₁), s)
s ∉ b  ⟹  (Some(R IF b THEN c₁ ELSE c₂ FI), s) →¹ (Some(c₂), s)

s ∉ b  ⟹  (Some(R WHILE  b INV inv DO c OD), s) →¹ (None, s)
s ∈ b  ⟹  (Some(R WHILE b INV inv DO c OD), s) →¹
           (Some(c;;inv WHILE b INV inv DO c OD), s)

⟦ s ∈ b; (Some c, s) →*ₐ (None, t) ⟧ ⟹
(Some(R AWAIT b THEN c END), s) →¹ (None, t)
```

Fig. 1. Transition rules for annotated commands

program depends on the variables that occur in it and their types. Hence the state is a parameter of the program type `ann_com`.

We follow [27] and model states as tuples. For example,

```
AnnBasic {(x,y). x=y} (λ(x,y). (x+1,y))
```

is the encoding of the assignment `x:=x+1` annotated with the precondition `x=y`. Fortunately, Isabelle's parser and printer can be extended with user-defined ML functions for translating between a nice external syntax and its internal representation. In the case of assignments, these functions translate between the external x `:=` e and the internal $\lambda(x_1,\ldots,x,\ldots,x_n).(x_1,\ldots,e,\ldots,x_n)$, where $x_1,\ldots,x_n$ is the list of all variables in the program, in some canonical order. Similarly, assertions have the external syntax $\{.P.\}$ (the dots avoid confusion with sets) which is turned into $\{(x_1,\ldots,x_n).\ P\}$. The translations are quite tricky because abstraction over tuples is not primitive in HOL but is realized by suitable combinations of ordinary abstraction and an uncurrying function of type $(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha \times \beta \Rightarrow \gamma$. To build the state space from the external syntax an explicit declaration of the variables is included.

4.4 Atomic Commands

Apt and Olderog [2] require the body of an `AWAIT` to be loop-free. This subclass of programs is realized in HOL by the type `atom_com`, which differs from `ann_com` in two important aspects:

- it contains neither `WHILE` nor `AWAIT`,

– it contains no annotations because, as the name indicates, `atom_com` is executed atomically, and hence there is no need to record a proof outline that can be checked for interference freedom.

Yet `atom_com` is sufficiently similar to `ann_com` that we do not show its syntax and operational semantics. Suffice it to say that $\rightarrow_a^*$ is the analogue of $\rightarrow^*$.

4.5 The Parallel Layer

The datatype `par_com` of parallel commands is defined like `ann_com`

```
α par_com = Parallel ((α ann_com option × α assn) list)
          | ParBasic (α ⇒ α)
          | ParSeq (α par_com) (α par_com)       ("_,, _")
          | ParCond (α bexp) (α par_com) (α par_com)
               ("IF _ THEN _ ELSE _ FI")
          | ParWhile (α bexp) (α assn) (α par_com)
               ("WHILE _ INV _ DO _ OD")
```

but with some differences:

– Parallel encloses a list of pairs (co, q), where co is an optional sequential command (or None, if the command has terminated), and q a postcondition (remember that the precondition is already part of the annotated co). Strictly speaking it is not necessary to include the postcondition, but it simplifies program verification. There is also some concrete syntax (not shown) of the form `COBEGIN` c_1 $\{.q_1.\}$ $\|$ ... $\|$ c_n $\{.q_n.\}$ `COEND`.
– The remaining commands are almost like their namesakes in the sequential layer, but with a slightly different concrete syntax, to avoid confusion. The only real difference is the missing precondition annotation. The latter is required to check for interference freedom. But, contrary to the name, there is no parallel execution of parallel commands (only of sequential commands), and hence no interference. `AWAIT` is superfluous for the same reason.

The operational semantics for `par_com` is a transition relation between pairs of parallel commands and the current state. The execution of the parallel composition of a list of sequential commands `Ts` proceeds by executing one non-None component of `Ts`:

$$[\![\ \texttt{i} \in \texttt{Index Ts};\ \texttt{Ts!i = (Some c, Q)};\ \texttt{(Some c, s)} \rightarrow^1 \texttt{(ro, t)}\ ]\!] \Longrightarrow$$
$$\texttt{(Parallel Ts, s)} \rightarrow_P^1 \texttt{(Parallel(Ts[i := (ro,Q)]), t)}$$

where Index `Ts` $\equiv$ {`i. i` $<$ length `Ts`}. This is the only rule for Parallel. We omit the rules for the remaining constructs because they are practically identical to the ones in the sequential layer. The only difference is that instead of None as a signal that execution of a parallel command has terminated we use Parallel `Ts`, where all commands of `Ts` are None. This is just a trick to avoid the `option` type. As a result, the first rule for sequential composition looks like this

$$\texttt{All\_None Ts} \Longrightarrow \texttt{((Parallel Ts,, c), s)} \rightarrow_P^1 \texttt{(c, s)}$$

where All_None `Ts` checks if the command-components of `Ts` are all None.

5 Proof Theory

The proof system for partial correctness of parallel programs is introduced in three stages, corresponding to the three layer hierarchy of the programming language. For each layer we show soundness of the proof system based on the soundness of the layer below. For the component layer we diverge most significantly from Apt and Olderog's treatment: having defined the operational semantics on a syntax including assertions, we do not need to formalize a notion of program locations.

Correctness formulas, also called *Hoare triples*, are statements of the form

$$\{P\}\ c\ \{Q\}$$

where c is a program and P and Q are the corresponding *precondition* and *postcondition*. *Partial correctness* means $\{P\}\ c\ \{Q\}$ is true iff every terminating computation of c that starts in a state s satisfying P ends in a state satisfying Q.

5.1 Proof System for Atomic Programs

Following Apt and Olderog, we define a *partial correctness semantics* of an atomic command $\mathtt{c}$ by

$$\mathsf{atom\_sem}\ \mathtt{c}\ \mathtt{s} \equiv \{\mathtt{t}.\ (\mathsf{Some}\ \mathtt{c},\ \mathtt{s}) \rightarrow_a^* (\mathsf{None},\ \mathtt{t})\}$$
$$\mathsf{atom\_SEM}\ \mathtt{c}\ \mathtt{S} \equiv \bigcup_{\mathtt{s}\in\mathtt{S}} \mathsf{atom\_sem}\ \mathtt{c}\ \mathtt{s}$$

where $\mathtt{s}$ is a state and $\mathtt{S}$ a set of states, and validity of a partial correctness formula by

$$(\models_a\ \mathtt{P}\ \mathtt{c}\ \mathtt{Q}) \equiv (\mathsf{atom\_SEM}\ \mathtt{c}\ \mathtt{P} \subseteq \mathtt{Q})$$

where $\mathtt{P}$ and $\mathtt{Q}$ are sets of states. Note that we employ the concrete assertion syntax $\{.P.\}$ introduced in §4.3 only in the case where P is a specific assertion referring to the variables of a specific program. Generic formulae involving assertions are more directly phrased in terms of variables like $\mathtt{P}$ ranging over sets of states.

The rules in the deductive system for atomic commands are the traditional ones. The notation $\vdash_a \mathtt{P}\ \mathtt{c}\ \mathtt{Q}$ stands for provability of a Hoare triple in this system.

Soundness of $\vdash_a$w.r.t. $\models_a$

$$\vdash_a\ \mathtt{P}\ \mathtt{c}\ \mathtt{Q} \Longrightarrow\ \models_a\ \mathtt{P}\ \mathtt{c}\ \mathtt{Q}$$

is proved by rule induction on the derivation of $\vdash_a \mathtt{P}\ \mathtt{c}\ \mathtt{Q}$. Given the following lemmas, all cases are automatically proved:

- $\mathsf{atom\_SEM}$ is monotonic,
- $\mathsf{atom\_SEM}\ (\mathtt{c}_1;;\mathtt{c}_2)\ \mathtt{S} = \mathsf{atom\_SEM}\ \mathtt{c}_2\ (\mathsf{atom\_SEM}\ \mathtt{c}_1\ \mathtt{S})$,
- $\mathsf{atom\_SEM}\ (\mathtt{IF}\ \mathtt{b}\ \mathtt{THEN}\ \mathtt{c}_1\ \mathtt{ELSE}\ \mathtt{c}_2\ \mathtt{FI})\ \mathtt{S} =$
  $\mathsf{atom\_SEM}\ \mathtt{c}_1\ (\mathtt{S} \cap \mathtt{b}) \cup \mathsf{atom\_SEM}\ \mathtt{c}_2\ (\mathtt{S} \cap (-\mathtt{b}))$.

5.2 Proof System for Component Programs

Proofs for component programs are presented in the form of standard proof outlines, where each command `c` is preceded by an assertion, `pre(c)`, and apart from these and loop invariants there are no other assertions. For purely sequential programs such a presentation is not necessary, since the weakest precondition can always be derived from the postcondition and loop invariants. However, in the parallel case this is sometimes insufficient to develop a proof; this situation can be remedied by explicitly stating stronger preconditions.

$$[\![\ \mathtt{P} \subseteq \{\mathtt{s.\ (f\ s)} \subseteq \mathtt{Q}\}\ ]\!] \Longrightarrow\ \vdash \mathtt{(AnnBasic\ P\ f)\ Q}$$

$$[\![\ \vdash \mathtt{c0\ pre(c1)};\ \vdash \mathtt{c1\ Q}\ ]\!] \Longrightarrow\ \vdash \mathtt{(c0;;c1)\ Q}$$

$$[\![\ \mathtt{(P} \cap \mathtt{b)} \subseteq \mathtt{pre(c1)};\ \vdash \mathtt{c1\ Q};\ \mathtt{(P} \cap \mathtt{(-b))} \subseteq \mathtt{pre(c2)};\ \vdash \mathtt{c2\ Q}\ ]\!]$$
$$\Longrightarrow\ \vdash \mathtt{(P\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI)\ Q}$$

$$[\![\ \mathtt{P} \subseteq \mathtt{inv};\ \mathtt{(inv} \cap \mathtt{b)} \subseteq \mathtt{pre(c)};\ \vdash \mathtt{c\ inv};\ \mathtt{(inv} \cap \mathtt{(-b))} \subseteq \mathtt{Q}\ ]\!]$$
$$\Longrightarrow\ \vdash \mathtt{(P\ WHILE\ b\ INV\ inv\ DO\ c\ OD)\ Q}$$

$$[\![\ \vdash_a \mathtt{(P} \cap \mathtt{b)\ c\ Q}\ ]\!] \Longrightarrow\ \vdash \mathtt{(P\ AWAIT\ b\ THEN\ c\ END)\ Q}$$

$$[\![\ \vdash \mathtt{c\ Q};\ \mathtt{Q} \subseteq \mathtt{Q'}\ ]\!] \Longrightarrow\ \vdash \mathtt{c\ Q'}$$

Fig. 2. Proof system for annotated commands

The formation rules for proof outlines are shown in Fig. 2. They look unusual because preconditions are hidden as part of the commands' syntax. Also note that the precondition of each command need not be the weakest one, so we can do without a rule to strengthen the precondition. The following theorem shows that this system is equivalent to the traditional presentation.

Theorem 1 (Equivalence of proof systems) Let $\vdash_{tr}$ `P` $\bar{\mathtt{c}}$ `Q` stand for provability of the correctness formula `P` $\bar{\mathtt{c}}$ `Q` in the traditional system. By $\bar{\mathtt{c}}$ we mean commands without any annotation other than loop invariants. The relation `c` $\sim \bar{\mathtt{c}}$ means that both commands are equal up to annotations (loop invariants must also be equal). Then,

(1) $\vdash_{tr} \mathtt{P}\ \bar{\mathtt{c}}\ \mathtt{Q} \Longrightarrow \exists\ \mathtt{c}.\ \vdash \mathtt{c\ Q} \land \mathtt{P} \subseteq \mathtt{pre(c)} \land \mathtt{c} \sim \bar{\mathtt{c}}$
(2) $\vdash \mathtt{c\ Q} \Longrightarrow \exists\ \bar{\mathtt{c}}.\ \vdash_{tr} \mathtt{pre(c)}\ \bar{\mathtt{c}}\ \mathtt{Q} \land \mathtt{c} \sim \bar{\mathtt{c}}$

The proofs are by rule induction on $\vdash_{tr}$ and $\vdash$ respectively.

Proof outlines enjoy the following intuitive property: 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 is called *strong soundness*.

Theorem 2 (Strong Soundness for Component Programs)
If $\vdash$ c Q and s $\in$ pre(c) and (Some c, s) $\rightarrow^*$ (ro, t) then

- if ro = Some r for a command r then t $\in$ pre(r),
- if ro = None then t $\in$ Q.

In particular for ro = None we have soundness in the usual way.

The proof is by rule induction on $\rightarrow^*$ with a nested induction on $\rightarrow^1$.

5.3 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. Thus, to conclude that the input/output specification of a list of component programs Ts executed in parallel is simply the intersection (assertions are modeled as sets) of the input/output specification of each component, we need not only study each component program proof independently, but must also guarantee that this proof is not falsified by the execution of any other component. This property, called *interference freedom of proof outlines*, is checked by INTERFREE Ts in the proof rule for parallel composition:

$$\llbracket\ \forall i \in \mathsf{Index}\ Ts.\ \exists\ c\ Q.\ Ts!i = (\mathsf{Some}\ c,\ Q) \wedge \vdash c\ Q\ ;\ \ \mathsf{INTERFREE}\ Ts\ \rrbracket \Longrightarrow$$

$$\vdash_P \bigcap_{i \in \mathsf{Index}\ Ts} \mathsf{pre}(\mathsf{the}(\mathsf{com}(Ts!i)))\ \ (\mathsf{Parallel}\ Ts)\ \bigcap_{i \in \mathsf{Index}\ Ts} \mathsf{post}(Ts!i)\ .$$

Remember that each element of Ts is a pair of an optional ann_com and an assn, the postcondition. Function post extracts the postcondition, com extracts the ann_com option, the extracts the command c from Some c (by assumption all commands are wrapped up in Some), and finally pre extracts the precondition.

The remaining proof rules of system $\vdash_P$ are standard and we concentrate on the formalization of INTERFREE, which requires a number of auxiliary concepts:

- An assertion P is invariant under execution of an atomic command r if $\models_a$ (P $\cap$ pre(r)) r P.
- We say that an atomic command r does not interfere with a standard proof outline $\vdash$ c Q iff the following two conditions hold:
  1. $\models_a$ (Q $\cap$ pre(r)) r Q,
  2. For any assertion P within c, $\models_a$ (P $\cap$ pre(r)) r P .
- Standard proof outlines $\vdash c_1\ Q_1, \ldots, \vdash c_n\ Q_n$ are interference free if no assignment or atomic region of a program c_i interferes with the proof outline $\vdash c_j\ Q_j$ of another program c_j, with $i \neq j$.

So, given two component programs (co,Q) and (co',Q'), showing interference freedom means proving that all assertions in the former are invariant under execution of all basic commands or atomic regions in the latter, and vice versa.

The test in one direction, which does not require the postcondition Q', is realized by function interfree:

```
interfree(co, Q, None) = True
interfree(None, Q, Some a) = ∀(R,r) ∈ atomics a. ⊨ₐ (Q ∩ R) r Q
interfree(Some c, Q, Some a) = ∀(R,r) ∈ atomics a. ⊨ₐ (Q ∩ R) r Q ∧
                                (∀ P ∈ assertions c. ⊨ₐ (P ∩ R) r P)
```

Atomic commands are extracted by the function atomics, which, given an annotated command, returns the set of all pairs (P, r) where r is either the body of an AWAIT-command, or a basic command AnnBasic R f, extracted as AtomBasic f, and P is the corresponding precondition. The set of all assertions of an annotated command (including loop invariants) is collected by the function assertions.

Function interfree must be applied to all possible combinations of component programs, except for a component program with itself:

```
INTERFREE Ts ≡ ∀ i ∈ Index Ts. ∀ j ∈ Index Ts.
                 i≠j ⟶ interfree (com(Ts!i), post(Ts!i), com(Ts!j))
```

With these preparations we can prove soundness of the proof rule for Parallel.

Theorem 3 (Strong Soundness for Parallel Composition)
Let Ts be a list of n component programs so that for each component (c_i, Q_i) the formula $\vdash c_i\ Q_i$ holds in the sense of partial correctness, and INTERFREE Ts holds. Suppose that (Parallel Ts, s) $\rightarrow_P^*$ (Parallel Rs, t) for some s $\in$ pre(c_i) for i $\in$ {1...n}, some list of component programs Rs and some state t. Let $Rs_j = (ro_j, Q_j)$ be the jth component of Rs. Then for j $\in$ {1...n},

- if ro_j = Some r_j for a command r_j then t $\in$ pre(r_j),
- if ro_j = None then t $\in$ Q_j.

In particular if ro_i = None for i $\in$ {1...n}, we have that t $\in$ Q_i for i $\in$ {1...n}.

The proof is by induction on the length of the computation. First we fix j $\in$ {1...n}. If the length is positive, we have the following situation:

(Parallel Ts, s) $\rightarrow_P^*$ (Parallel Rs, b) $\rightarrow_P^1$ (Parallel(Rs[i:=(ro_i, Q_i)]), t)

where the last step was performed by the ith component of Rs, (Some l_i, Q_i), through transition (Some l_i, b) $\rightarrow^1$ (ro_i, t).

Now two cases arise: i=j and i≠j. The first one is resolved with help of the Strong Soundness of Component Programs Theorem. The second one deserves some more attention.

It is proved by rule induction on the relation $\rightarrow^1$. It turns out that the induction hypothesis is too weak to solve both sequential cases. Therefore we need to prove a stronger lemma whose conclusion contains one more clause than the theorem above, namely INTERFREE Rs. This means that interference freedom is preserved throughout the computation, which is easily proved. Now all cases for i≠j can be solved and the proof is complete.

Soundness of the remaining inference rules for $\vdash_P$ is analogous to the case of atomic programs, with two main differences:

– Because of the lack of None as sign of termination, par_sem is now

  par_sem c s $\equiv$ {t. $\exists$ Ts. (c,s) $\rightarrow_P^*$ (Parallel Ts, t) $\wedge$ All_None Ts}

– We also have to consider the rule for while loops, whose proof of soundness relies on the following lemma:

  par_SEM (WHILE b INV inv DO c OD) = $\bigcup_{k=0}^{\infty}$ par_SEM (fwhile b c k)

  fwhile is defined by induction as the following sequence of programs

```
fwhile b c 0 = Omega
fwhile b c (Suc n) = IF b THEN c,,(fwhile b c n) ELSE SKIP FI
```

  where Omega and SKIP are the abbreviations

```
SKIP ≡ ParBasic Id
Omega ≡ WHILE TRUE INV TRUE DO SKIP OD .
```

5.4 Verification condition generation

The proof rules of our systems 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. Due to lack of space, we cannot go into details. Suffice it to say that, although the overall structure of the tactic is straightforward, there are some tricky issues involved because the state space is not fixed but depends on the number of variables that occur in the program. Thus we need to prove on the fly special instances of the assignment axiom that are specific to the program under consideration.

6 Examples

We have verified all the relevant examples in [2] and some others taken from the literature. Due to lack of space, we only present the producer/consumer problem (in the first subsection); a second subsection looks at a simple example of a schematic program where the number of components is not fixed.

The examples rely heavily on arrays which are modeled as lists. Thus array access `A[i]` becomes `A!i` and array update `A[i]:=e` becomes `A:=A[i:=e]`.

6.1 Producer/Consumer

This problem models two processes, producer and consumer, that share a common, bounded buffer. The producer puts information into the buffer, the consumer takes it out. Trouble arise when the producer attemps to put a new item in a full buffer or the consumer tries to remove an item from an empty buffer. Following Owicki and Gries we express the problem as a parallel program with shared variables and `AWAIT`-commands. It copies an array `a` into an array `b`

```
{.0 < M₁ ∧ 0 < N ∧ M₁ = M₂.}
COBEGIN
 PROD || CONS
COEND
{.∀ k. k < M₁ ⟶ a!k = b!k.}
```

where `N = length buffer`, M_1 = length `a` and M_2 = length `b`. 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 in Fig. 3.

Both components share the variables `in` and `out`, which count the values added to the buffer and the values removed from the buffer, respectively. Thus, the buffer contains `in-out` values at each moment. Expressions `in mod N` and `out mod N` determine the subscript of the buffer element where the next value is to be added or removed.

For readability we use following abbreviations:

```
INIT = 0 < M₁ ∧ 0 < N ∧ M₁ = M₂
I = ∀ k. (out ≤ k ∧ k < in ⟶ a!k = buffer!(k mod N) )
    ∧ out ≤ in ∧ in-out ≤ N
I₁ = I ∧ i ≤ M₁
p₁ = I₁ ∧ i = in
I₂ = I ∧ (∀ k. k < j ⟶ a!k = b!k) ∧ j ≤ M₁
p₂ = I₂ ∧ j = out
```

The command `WAIT b END` abbreviates `r AWAIT b THEN SKIPEND`.

The verification of this problem involves proving a total of 88 conditions. Half of them are trivially solved since they refer to triples of the form `(A ∩ pre(r)) r A` where the atomic action `r` does not change the variables in `A`. The rest are automatically solved by an Isabelle simplification tactic.

6.2 Schematic programs

So far we have only considered programs with a fixed number of parallel components, because that is all the programming language allows. However, we would also like to establish the correctness of program schemas such as

```
COBEGIN A := A[1 := 0] || ... || A := A[n := 0] COEND
```

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`) we can express the above schematic program in HOL as follows

```
Parallel (map (λ i. {.i < length A.} A := A[i:=0] {.A!i = 0.}) [1..n])
```

```
⊢_P vars: in out i j x y buffer b.
{.INIT.}
in:=0,, out:=0,, i:=0,, j:=0,,
COBEGIN
    {.p1 ∧ INIT.}
    WHILE i < M1  INV {.p1 ∧ INIT.}
    DO {.p1 ∧ i < M1 ∧ INIT.}
       x:= a!i;;
       {.p1 ∧ i < M1 ∧ x = a!i ∧ INIT.}
       WAIT in-out < N END;;
       {.p1 ∧ i < M1 ∧ x = a!i ∧ in-out < N ∧ INIT.}
       buffer:= buffer[(in mod N):= x];;
       {.p1 ∧ i < M1 ∧ a!i = buffer!(in mod N) ∧ in-out < N ∧ INIT.}
       in:= in+1;;
       {.I1 ∧ (i+1) = in ∧ i < M1 ∧ INIT.}
       i:= i+1
    OD
    {.p1 ∧ i = M1 ∧ INIT.}
‖
    {.p2 ∧ INIT.}
    WHILE j < M1  INV {.p2 ∧ INIT.}
    DO {.p2 ∧ j < M1 ∧ INIT.}
       WAIT out < in END;;
       {.p2 ∧ j < M1 ∧ out < in ∧ INIT.}
       y:= buffer!(out mod N);;
       {.p2 ∧ j < M1 ∧ out < in ∧ y = a!j ∧ INIT.}
       out:= out+1;;
       {.I2 ∧ (j+1) = out ∧ j < M1 ∧ y = a!j ∧ INIT.}
       b:= b[j:=y];;
       {.I2 ∧ (j+1) = out ∧ j < M1 ∧ a!j = b!j ∧ INIT.}
       j:= j+1
    OD
    {.p2 ∧ j = M1 ∧ INIT.}
COEND
{.∀ k. k<M1 ⟶ a!k = b!k.}
```

Fig. 3. Producer/Consumer Problem

where we have already inserted the necessary annotations to prove

```
⊢_P {.n < length A.} Parallel(...) {.∀ i. 1≤i ∧ i≤n ⟶ A!i = 0.} .
```

Suffice it to say that the proof is fairly straightforward and does not even require induction on n, as one may have expected. We do not present the details because they are neither intellectually challenging nor very readable. The problem is that currently both pretty printing and verification condition generation only works for fully instantiated programs but not for schemas. Hence proofs about schematic programs are still more tedious than they could be.

7 Conclusion

We have presented the first formalization and (an improved) soundness proof of the Owicki/Gries method in a general purpose theorem prover. This is another step towards the embedding of realistic imperative programming languages and their verification calculi in a theorem prover. So far we have concentrated on the logical infrastructure and typical examples from the literature. We intend to extend our work in two directions: formalization of a compositional proof system and support for schematic programs as in §6.2. A completeness proof would also be interesting, but this is not our main focus. However, it should be mentioned that our current proof system is incomplete because there is no rule for removing auxiliary variables, an omission that is easy to fix.

The whole development required roughly three quarters of a person year: half a year to produce the first version, and another quarter of a year to polish the theory and document it. Most of the time was consumed by exploring different formalizations. The complete formalization comprises 350 lines of definitions, 1150 lines of proofs and 550 lines of ML code (for the parser and printer translation functions and the tactic explained in §5.4).

Acknowledgment We thank Javier Esparza, David von Oheimb, Cornelia Pusch and Markus Wenzel for the very helpful discussions, and two anonymous referees for their comments.

References

1. 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.
2. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.
3. A. Camillieri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16:993–1004, 1990.
4. 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.
5. F. de Boer, U. Hannemann, and W.-P. de Roever. A compositional proof system for shared variable concurrency. 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 515–532. Springer-Verlag, 1997.
6. 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.

7. 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.
8. D. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16:1005–1022, 1990.
9. 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.
10. M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
11. 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.
12. Isabelle home page. `www.cl.cam.ac.uk/Research/HVG/isabelle.html`.
13. C. B. Jones. Development methods for computer programs including a notion of interference. Technical Report PRG-25, Programming Research Group, Oxford University Computing Laboratory, 1981.
14. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Programming Languages and Systems*, 5:596–619, 1983.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. T. Nipkow. *Isabelle/HOL. The Tutorial*, 1998. Unpublished Manuscript. Available at `www.in.tum.de/~nipkow/pubs/HOL.html`.
21. T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, ?, 1998. To appear.
22. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
23. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
24. J. Søgaard-Andersen, S. Garland, J. Guttag, N. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In *Fourth Conference on Computer-Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 305–319. Springer-Verlag, 1993.

25. K. Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, Computer Science Department, Manchester University, 1990.
26. 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.
27. J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.
28. 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.