The Rely-Guarantee Method in Isabelle/HOL

Leonor Prensa Nieto

INRIA Sophia-Antipolis, France[†]
Leonor.Prensa@inria.fr

Abstract We present the formalization of the rely-guarantee method in the theorem prover Isabelle/HOL. This method consists of a Hoare-like system of rules to verify concurrent imperative programs with shared variables in a compositional way. Syntax, semantics and proof rules are defined in higher-order logic. Soundness of the proof rules w.r.t. the semantics is proven mechanically. Also parameterized programs, where the number of parallel components is a parameter, are included in the programming language and thus can be verified directly in the system. We prove that the system is complete for parameterized programs. Finally, we show by an example how the formalization can be used for verifying concrete programs.

1 Introduction

The rely-guarantee method introduced by Jones [5] represents the first and most fundamental compositional method for correctness proofs of parallel imperative programs with shared variables. It consists of a set of axioms and inference rules that form a sound and complete system for the derivation of correct programs in the style of Hoare. It also has the classical advantages of Hoare logic, namely, it is syntax oriented and compositional. In a compositional proof system, the specification of a parallel program can be derived from the specification of the components without knowing the implementation of these components. This is important for the correct development of complex programs, where one would like to verify the design at stages where implementation details are still unknown.

The rely-guarantee method can be considered as a reformulation of the classical non-compositional Owicki-Gries method (also formalized in Isabelle/HOL [8]). To apply the Owicki-Gries method, programs have to be annotated with an assertion at every point of interference. The verification process requires proving that the annotations of each component be preserved by the atomic actions of the other components. This property, called *interference freedom*, makes the method non compositional because the particular implementation of the components must be known. The idea of the rely-guarantee method is to record the interference information in the specification of each component. Hence, besides the classical *pre* and *postcondition* of Hoare logic, each component is annotated with a *rely* and a *guarantee* condition, which describe the expected effect of

[†] This work has been realized as part of the author's PhD at the Computer Science Faculty of the Technical University of Munich.

the environment and of the component itself, respectively. Then, the verification process requires proving correctness of each component separately and some side conditions about their specifications, for which no knowledge of the internal implementation of the components is required. That is, the resulting proof method is compositional.

This paper presents the formalization in the theorem prover Isabelle/HOL of the rely-guarantee proof system. The main results are:

– A higher-order logic model of: parallel programs, a semantic definition of correctness and a proof system.
– A formalized theorem that the proof system is sound w.r.t. the semantics.

An interesting by-product of our formalization is that parameterized programs, where the number of components is a parameter n, are naturally included in the model. This is a consequence of the representation of parallel programs as lists of components. Our proof rule for parallel composition allows us to derive correct specifications of parameterized programs directly, without induction. A soundness and completeness proof for such a system is new in the literature.

Finally, we show by an example how the formalization can be used to verify concrete programs. In practice, the real challenge is to identify suitable rely and guarantee conditions. This requires a full understanding of the program and a detailed identification of the interactions that occur. Such verification exercises are tedious and error prone. A theorem prover is a great help in the iterative process of finding and adjusting the specifications; previous proofs can be easily reused and details are checked automatically. The user can then concentrate only on the most interesting steps.

The definitions and theorems shown in the paper are actual fragments of the Isabelle theories and we hope to convince the reader of the expressiveness of Isabelle's syntax. Due to lack of space we cannot show all the definitions and proofs. For a detailed exposition we refer to [11]. The full theories and proof scripts are available at http://isabelle.in.tum.de/library/HOL/HoareParallel/.

2 Related Work

The formalization presented here is mostly inspired by [15], where the system is proved to be sound and complete. The preciseness required by a theorem prover, however, leads to some simplifications and improvements over the original model. There exists a broad literature on the rely-guarantee and other related systems that we cannot survey here. The recent book [1] presents systematically and in a unified notation the work of more than a 100 publications and 15 dissertations on concurrency verification.

From the theorem prover angle, much work has been done on formalizing different concurrency paradigms like UNITY, CSP, CCS or TLA among others (see [8] for a list of references). Remarkable formalizations for compositional approaches are [2, 9] for the UNITY framework in Isabelle/HOL and the soundness proof of McMillan assume-guarantee rule [6] in PVS [12]. Surprisingly, there is not much work on embedding Hoare logics for parallelism in theorem provers. A Hoare-style compositional proof system for distributed real-time systems has been formalized and proved correct in PVS [4]. In this formalization, variables are local, i.e. not shared by parallel components and communication is achieved by means of events which are then used to model different forms of communication. Also a static checker for parallel programs has been presented in [3], where given a suitable rely-guarantee specification for a parallel program, the tool decomposes it in a verification problem for the sequential components, which can be checked by an automatic theorem prover. This tool is focused on the verification and no soundness proof has been formalized. To the best of our knowledge, the work presented in this paper is the first formalization in a theorem prover of a compositional system and its soundness proof for shared-variable parallelism in the style of Hoare.

3 Isabelle/HOL

Isabelle is a generic interactive theorem prover and Isabelle/HOL is its instantiation for higher-order logic. For a gentle introduction to Isabelle/HOL see [7]. Here we summarize the relevant notation and some predefined functions used in the paper. Others will be explained as they appear.

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 $xs\,!\,i$. The last element of a non-empty list is *last xs*. The functional $map :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta\ list$ applies a function to all elements of a list. The syntax $xs[i := x]$ denotes the list xs with the ith component replaced by x.

The datatype $\alpha\ option = None \mid Some\ \alpha$ is frequently used to add a distinguished element to some existing type. It comes with the function *the* such that *the (Some x)* $= x$. Set comprehension syntax is $\{x.\ P\ x\}$ expressing the set of all elements that satisfy the predicate P. The complement of a set A is $-A$. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ represents an implication with assumptions $A_1, \ldots, A_n$ and conclusion A.

4 The Programming Language

We formalize a simple while-language augmented with shared-variable parallelism ($\|$) and synchronization via an await-construct. For simplicity, each P_i in **cobegin** $P_1 \| \ldots \| P_n$ **coend** must be a sequential command, i.e. nested parallelism is not allowed. We encode this stratification by defining the syntax in two layers, one for sequential component programs and another for parallel programs. We start by defining boolean expressions as sets of states, where the state is represented by the parameter α:

types α *bexp* $= \alpha$ *set*

The syntax of component programs is given by the following datatype:

$$
\begin{array}{lll}
\textbf{datatype}\ \alpha\ com = & Basic\ \ (\alpha \Rightarrow \alpha) & \\
& \mid\ Seq\ \ (\alpha\ com)\ \ (\alpha\ com) & (\text{-; -}) \\
& \mid\ Cond\ \ (\alpha\ bexp)\ \ (\alpha\ com)\ \ (\alpha\ com) & \\
& \mid\ While\ \ (\alpha\ bexp)\ \ (\alpha\ com) & \\
& \mid\ Await\ \ (\alpha\ bexp)\ \ (\alpha\ com) &
\end{array}
$$

The *Basic* command represents an atomic state transformation, for example, an assignment, a multiple assignment, or the *skip* command. The *Await* command executes the body atomically whenever the boolean condition holds. The rest are well-known. Parallel programs, on the other layer, are simply lists of component programs:

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

The option type is used to include the empty program *None* as a possible component program.

For the moment, we only introduce concrete syntax of the form $c_1;\ c_2$ for sequential statements. Concrete syntax is nice for representing and proving properties of concrete programs. The main difficulty for defining concrete syntax lies in finding a convenient representation of the state, or more precisely, of program variables. We will come back to this issue in the example of section 10. The rest of the paper, however, deals with meta-theory, i.e. definitions and proofs about the language itself, so we use the abstract syntax and leave the state completely undetermined.

5 Operational Semantics

Semantics of commands is defined via transition rules between configurations. A configuration is a pair $(P,\ \sigma)$, where P is some program (or the empty program) and σ is a state. A transition rule has the form $(P,\ \sigma)\ {-\delta\rightarrow}\ (P',\ \sigma')$ where δ is a label indicating the kind of transition. A component program can perform two kinds of transitions: *component transitions* (labeled with c), performed by the component itself, and *environment transitions* (labeled with e), performed by a different component of the parallel composition or by an arbitrary environment.

5.1 Transition Rules

Rules for Component Programs: The rule for environment transitions is

Env: $(P,\ s)\ {-e\!\longrightarrow}\ (P,\ t)$

Intuitively, a transition made by the environment of a component program P may change the state but not the program P. The program part is only modified by transitions made by the component itself. Such transitions are inductively defined by the following rules:

$Basic$: $(Some\ (Basic\ f),\ s)\ -c\rightarrow\ (None,\ f\ s)$

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

$Seq1$: $(Some\ P_0,\ s)\ -c\rightarrow\ (None,\ t) \Longrightarrow (Some\ (P_0;\ P_1),\ s)\ -c\rightarrow\ (Some\ P_1,\ t)$
$Seq2$: $(Some\ P_0,\ s)\ -c\rightarrow\ (Some\ P_2,\ t)$
$\Longrightarrow (Some\ (P_0;\ P_1),\ s)\ -c\rightarrow\ (Some\ (P_2;\ P_1),\ t)$

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

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

where $P\ -c{*}\rightarrow\ Q$ is the reflexive transitive closure of $P\ -c\rightarrow\ Q$. Basic actions and evaluation of boolean conditions are atomic. The body of an await-statement is executed atomically, i.e. without interruption from the environment, thus no $e$nvironment transitions can occur.

Rules for Parallel Programs: Parallel programs may also interact with the environment, thus an analogous environment transition, labeled with pe, is defined:

$ParEnv$: $(Ps,\ s)\ -pe\rightarrow\ (Ps,\ t)$

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:

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

$Ps[i{:=}r]$ is the list of programs Ps with the program i replaced by r. A parallel program terminates when all the components terminate, i.e. when all component programs are $None$.

5.2 Computations

A *computation* of a sequential program records the sequence of transitions. In [15] it is defined as any sequence of the form

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

There are several ways to formalize this intuitive definition. We present two formalizations that are equivalent but serve different purposes. The first one directly follows the definition and is "obviously" the right one. The second one is more elaborated and is useful for the proofs.

Direct Definition of Computation: We define the set of computations, called *cptn*, as the set of lists of configurations inductively defined by the following rules:

One: $[(P,\ s)] \in cptn$
Env: $(P,\ t)\#xs \in cptn \Longrightarrow (P,\ s)\#(P,\ t)\#xs \in cptn$
$Comp$: $[\![\ (P,\ s)\ -c\rightarrow\ (Q,\ t);\ (Q,\ t)\#xs \in cptn\ ]\!] \Longrightarrow (P,\ s)\#(Q,\ t)\#xs \in cptn$

The one-element list is always a computation. Two consecutive configurations are part of a computation if they are the initial and final configurations of an environment or a component transition. Computations of parallel programs (*par-cptn*) are defined analogously.

Modular Definition of Computation: The previous definition of computation clearly formalizes the one proposed in [15]. However, it represents the execution of a program in a simplified linear way without taking the structure of the development of a computation into account. For example, the computation of a sequential composition is formed by the computation of the two parts and the computation of a while-statement is formed by several computations of the body. Retrieving this information out of the linear representation of the computation is unnecessarily cumbersome. It can be elegantly avoided by defining computations in a modular way. We propose a new definition of computation which maintains the structure and considerably simplifies some proofs, especially those concerning properties of while-programs.

First, we define the auxiliary function *seq-with* that returns, given a program Q and a configuration $(P,\ s)$, 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[3]:

$$seq\text{-}with\ Q \equiv \lambda(P,s).\ if\ P{=}None\ then\ (Some\ Q,\ s)\ else\ (Some((the\ P);\ Q),\ s)$$

We define the set of computations *mcptn* as the lists of configurations formed by the following rules:

$MOne$: $[(P,\ s)] \in mcptn$

$MEnv$: $(P,\ t)\#xs \in mcptn \Longrightarrow (P,\ s)\#(P,\ t)\#xs \in mcptn$

$MNone$: $[\![\ (Some\ P,\ s)\ -c\rightarrow\ (None,\ t);\ (None,\ t)\#xs \in mcptn\ ]\!]$
$\Longrightarrow (Some\ P,\ s)\#(None,\ t)\#xs \in mcptn$

$MCondT$: $[\![\ (Some\ P_0,\ s)\#ys \in mcptn;\ s \in b\ ]\!]$
$\Longrightarrow (Some\ (Cond\ b\ P_0\ P_1),\ s)\#(Some\ P_0,\ s)\#ys \in mcptn$

$MCondF$: $[\![\ (Some\ P_1,\ s)\#ys \in mcptn;\ s \notin b\ ]\!]$
$\Longrightarrow (Some\ (Cond\ b\ P_0\ P_1),\ s)\#(Some\ P_1,\ s)\#ys \in mcptn$

[3] Isabelle's notation does not allow tuples as arguments on the left-hand side of a definition. Thus, λ-notation is used on the right-hand side.

$MSeq1$: $[\![$ $(Some\ P_0,\ s)\#xs \in mcptn;\ zs = map\ (seq\text{-}with\ P_1)\ xs$ $]\!]$
$\Longrightarrow (Some\ (P_0;\ P_1),\ s)\#zs \in mcptn$

$MSeq2$: $[\![$ $(Some\ P_0,\ s)\#xs \in mcptn;\ fst\ (last\ ((Some\ P_0,\ s)\#xs)) = None;$
$(Some\ P_1,\ snd\ (last\ ((Some\ P_0,\ s)\#xs)))\#ys \in mcptn;$
$zs = (map\ (seq\text{-}with\ P_1)\ xs)@ys$ $]\!]$ $\Longrightarrow (Some\ (P_0;\ P_1),\ s)\#zs \in mcptn$

$MWhile1$: $[\![$ $(Some\ P,\ s)\#xs \in mcptn;\ s \in b;\ zs = map\ (seq\text{-}with\ (While\ b\ P))\ xs$ $]\!]$
$\Longrightarrow (Some\ (While\ b\ P),\ s)\#(Some\ (P;\ While\ b\ P),\ s)\#zs \in mcptn$

$MWhile2$: $[\![$ $(Some\ P,\ s)\#xs \in mcptn;\ fst\ (last\ ((Some\ P,\ s)\#xs)) = None;$
$s \in b;\ zs = (map\ (seq\text{-}with\ (While\ b\ P))\ xs)@ys;$
$(Some\ (While\ b\ P),\ snd\ (last\ ((Some\ P,\ s)\#xs)))\#ys \in mcptn$ $]\!]$
$\Longrightarrow (Some\ (While\ b\ P),\ s)\#(Some\ (P;\ While\ b\ P),\ s)\#zs \in mcptn$

The first two rules are the same as in the set or rules defining *cptn*. The rule *Comp*, however, is now replaced by seven rules which correspond to different kinds of component transitions.

The rule *MNone* stands for the three possible component transitions where the program terminates, i.e. *Basic*, *Await* or *WhileF*. 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 rule *Comp* with case analysis on the corresponding *c*-step.) Rule $MSeq1$ represents the case where the second program of the sequential composition is not started, whereas $MSeq2$ stands for the case where at least the first program is finished. $MWhile1$ represents the computations where the body is started but not finished and $MWhile2$ those where the body has been executed at least once.

The new definition is specially useful for the proof of soundness of the rule for while-programs. By using rule induction on *mcptn* we directly obtain the three following cases:

1. The while-body is not entered.
2. The execution of the body is at least started.
3. The body is executed completely at least once followed by a new computation of the same while-program, on which the induction hypothesis holds.

In contrast, the information obtained by using the same proof method on *cptn* was almost useless. The equivalence of both definitions is proven in the following theorem:

theorem *cptn-iff-mcptn*: $cptn = mcptn$

6 Validity of Correctness Formulas

In this section we formally define what it means for a program P to satisfy a rely-guarantee specification (*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 transitions from the environment and transitions of the program, respectively. Thus, they are sets of pairs of states, formed by the state before and after the transition.

P satisfies its specification, written $\models P$ **sat** $[pre, rely, guar, post]$, 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*.

Formally, validity of a specification for a sequential component program is defined as follows:

$\models P$ **sat** $[pre,\ rely,\ guar,\ post] \equiv \forall s.\ cp\ (Some\ P)\ s \cap assum\ (pre,\ rely) \subseteq comm\ (guar,\ post)$

where $cp\ (Some\ P)\ s$ represents the set of computations of the component program P starting from some initial state s, i.e. $cp\ (Some\ P)\ s \equiv \{c.\ c!0 = (Some\ P,\ s) \wedge c \in cptn\}$. The definitions of *assum* and *comm* are:

$$\begin{aligned}
assum \equiv \lambda(pre,\ rely).\ \{c.\ & snd\ (c!0) \in pre \wedge (\forall\, i.\ i{+}1 < length\ c \longrightarrow \\
& c!i \ {-e\rightarrow}\ c!(i{+}1) \longrightarrow (snd\ (c!i),\ snd\ (c!(i{+}1))) \in rely)\} \\
comm \equiv \lambda(guar,\ post).\ \{c.\ & (\forall\, i.\ i{+}1 < length\ c \longrightarrow \\
& c!i \ {-c\rightarrow}\ c!(i{+}1) \longrightarrow (snd\ (c!i),\ snd\ (c!(i{+}1))) \in guar) \wedge \\
& (fst\ (last\ c) = None \longrightarrow snd\ (last\ c) \in post)\}
\end{aligned}$$

In other words, P satisfies its specification iff all computations of P that satisfy the assumptions satisfy the commitments.

Validity of a specification of a parallel program Ps (of type α *par-com*), written $\Vdash Ps$ **sat** $[pre,\ rely,\ guar,\ post]$, is defined analogously. (Note the syntactic difference between $\models$ used for component programs and $\Vdash$ for parallel programs.)

Jones [5] first suggested that the rely and guarantee conditions be reflexive and transitive. However, for the soundness proof only the reflexivity of the guarantee condition is necessary. This is to ensure that transitions corresponding to the evaluation of boolean conditions (which do not affect the state) also satisfy the guarantee condition. If transitivity is also required another property, namely, observational equivalence, can be proven. In [14], the author discusses this point in more detail. Similarly, he requires only reflexivity since in practice finding guarantee conditions that are transitive is not easy.

7 The Proof System

First, we define *stable* $p\ q \equiv \forall\, x\ y.\ x \in p \longrightarrow (x,\ y) \in q \longrightarrow y \in p$. Thus, *stable pre rely* reads as "pre is stable when rely holds" meaning that if a state from the precondition performs a transition satisfying the rely condition, then the next state still satisfies the precondition.

The derivable correctness formulas $\vdash P$ **sat** $[pre,\ rely,\ guar,\ post]$ are inductively defined by the following set of rules:

$$\begin{aligned}&\mathit{Basic}\text{: } [\![\ pre \subseteq \{s.\ f\ s \in post\};\ \{(s,\ t).\ s \in pre \wedge (t = f\ s \vee t = s)\} \subseteq guar;\\ &\quad stable\ pre\ rely;\ stable\ post\ rely\ ]\!] \Longrightarrow \vdash \mathit{Basic}\ f\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]\end{aligned}$$

$$\begin{aligned}&\mathit{Seq}\text{: } [\![\vdash P\ \textsf{sat}\ [pre,\ rely,\ guar,\ mid];\ \vdash Q\ \textsf{sat}\ [mid,\ rely,\ guar,\ post]\ ]\!]\\ &\quad \Longrightarrow \vdash P;\ Q\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]\end{aligned}$$

$$\begin{aligned}&\mathit{Cond}\text{: } [\![\vdash P_1\ \textsf{sat}\ [pre \cap b,\ rely,\ guar,\ post];\ \vdash P_2\ \textsf{sat}\ [pre \cap -b,\ rely,\ guar,\ post];\\ &\quad stable\ pre\ rely;\ \forall\, s.\ (s,\ s) \in guar\ ]\!] \Longrightarrow \vdash \mathit{Cond}\ b\ P_1\ P_2\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]\end{aligned}$$

$$\begin{aligned}&\mathit{While}\text{: } [\![\vdash P\ \textsf{sat}\ [pre \cap b,\ rely,\ guar,\ pre];\ pre \cap -b \subseteq post;\ stable\ post\ rely;\\ &\quad stable\ pre\ rely;\ \forall\, s.\ (s,\ s) \in guar\ ]\!] \Longrightarrow \vdash \mathit{While}\ b\ P\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]\end{aligned}$$

$$\begin{aligned}&\mathit{Await}\text{: } [\![\ \forall\, V. \vdash P\ \textsf{sat}\ [pre \cap b \cap \{V\},\ \{(s,\ t).\ s = t\},\ UNIV,\\ &\quad \{s.\ (V,\ s) \in guar\} \cap post];\ stable\ pre\ rely;\ stable\ post\ rely\ ]\!]\\ &\quad \Longrightarrow \vdash \mathit{Await}\ b\ P\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]\end{aligned}$$

$$\begin{aligned}&\mathit{Conseq}\text{: } [\![\ pre \subseteq pre';\ rely \subseteq rely';\ guar' \subseteq guar;\ post' \subseteq post;\\ &\quad \vdash P\ \textsf{sat}\ [pre',\ rely',\ guar',\ post']\ ]\!] \Longrightarrow \vdash P\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]\end{aligned}$$

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. 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.

The rule for the await-statement is less obvious. 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*, defined as $\{s.\ True\}$. 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, *stable post rely* is required. Finally, the rule of consequence allows us to strengthen the assumptions and weaken the commitments.

We now introduce the proof rule for parallel composition. Recall that in a validity formula for a parallel program $\models Ps$ **sat** $[pre,\ rely,\ guar,\ post]$, Ps has type α *par-com* with no information about the *pre*, *post*, *rely* and *guar* conditions of each component program. This is fine to define validity, however, for concrete verification of programs, we want to apply the rules backwards. Therefore, the conclusion of the rule should include all the information needed in the premises. Hence, in a derived formula for a parallel program, denoted $\Vdash Ps$ **sat** $[pre,\ rely,\ guar,\ post]$, Ps is a list of tuples, each one formed by the code of the component program and its specification. The functions *Pre*, *Post*, *Rely*, *Guar* and *Com* (with obvious definitions) extract the different parts when applied to such a "component tuple".

Parallel:

$$\begin{array}{l}
[\![\ \forall i{<}length\ Ps.\ \vdash Com(Ps!i)\ \textsf{sat}\ [Pre(Ps!i),\ Rely(Ps!i),\ Guar(Ps!i),\ Post(Ps!i)];\\
\ \ \forall i{<}length\ Ps.\ rely \cup (\bigcup j{\in}\{j.\ j{<}length\ Ps \wedge j{\neq}i\}.\ Guar\ (Ps!j)) \subseteq Rely\ (Ps!i);\\
\ \ (\bigcup j{\in}\{j.\ j{<}length\ Ps\}.\ Guar\ (Ps!j)) \subseteq guar;\\
\ \ pre \subseteq (\bigcap i{\in}\{i.\ i{<}length\ Ps\}.\ Pre\ (Ps!i));\\
\ \ (\bigcap i{\in}\{i.\ i{<}length\ Ps\}.\ Post\ (Ps!i)) \subseteq post\ ]\!] \Longrightarrow\ \Vdash Ps\ \textsf{sat}\ [pre,\ rely,\ guar,\ post]
\end{array}$$

We explain the five premises. The first one requires that each component together with its specification be derivable in the system for sequential program.

The second one is a constraint on the rely condition of component i. An environment transition for i corresponds to a component transition of another component j with $i{\neq}j$, or of a transition from the overall environment (which satisfies *rely*). Hence, the strongest rely condition for component i is $rely \cup (\bigcup j{\in}\{j.\ j{<}length\ Ps \wedge j{\neq}i\}.\ Guar\ (Ps!j))$.

The third requirement imposes a relation among the guarantee conditions of the components and that of the parallel composition: since a component transition of the parallel program is performed by one of its components, the guarantee condition *guar* of the parallel program must be at least the union of the sets specified by the guarantee conditions of the components.

The forth premise requires that the precondition for the parallel composition imply all the component's preconditions. Finally, the overall postcondition must be a logical consequence of all postconditions.

This rule generalizes the particular case of composing two programs, as known from the literature [1, 15], to composing any number of programs. As a consequence, also parameterized parallel programs can be proved correct in a single derivation even though they represent an infinite family of programs (see section 9).

8 Soundness

To prove soundness of the rule for parallel composition we first need to prove soundness of the system of rules for sequential component programs:

theorem *sound*: $\vdash P$ **sat** $[pre,\ rely,\ guar,\ post] \Longrightarrow \models P$ **sat** $[pre,\ rely,\ guar,\ post]$

To state soundness of the parallel composition rule, we define a function *ParallelCom* which, given a list of "component tuples" formed by each component program's code and its corresponding four conditions it returns the same list with only the component programs, i.e. the parallel program. The soundness theorem is formulated using this function as follows:

theorem *par-sound*:
$\Vdash Ps$ **sat** $[pre,\ rely,\ guar,\ post] \Longrightarrow \models\!\models$ *ParallelCom Ps* **sat** $[pre,\ rely,\ guar,\ post]$

Both proofs are done by rule induction. For the soundness of the system for component programs the most interesting case is the rule for while, where the use of the modular definition of computation results in an elegant and well-structured proof. Soundness of the parallel rule relies on an important lemma stating that the computation of a parallel program can be described in terms of the computations of its components, i.e. that the semantics is compositional. This result alone has the longest proof (about 500 lines).

9 Completeness for Parameterized Parallel Programs

By using lists to model parallel composition (see section 4) we can easily represent parameterized parallel programs via the predefined HOL functional *map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta\ list$, and the construct $[i..j]$, which represents the list of natural numbers from i to j. For example, the program **cobegin** $P\ 0 \parallel \ldots \parallel P\ (n{-}1)$ **coend** representing the parallel composition of n components which differ only on the index number can be represented by $map\ (\lambda i.\ P\ i)\ [0..n{-}1]$, for which concrete syntax of the form **scheme** $[0 \leq i < n]\ P\ i$ has also been defined.

Consequently, the rule for parallel composition and its soundness proof also include the case of parameterized parallel programs. This means that correctness for such programs can be proven by a single derivation (typically parameterized by the number of components) which can then be instantiated for any parameter. This leads to the question whether finding such a derivation is always possible, i.e. whether the system is complete for this kind of programs.

In [10] we proved that the extended Owicki-Gries system of [8] is complete for parameterized programs. Using this result, we prove completeness of the rely-guarantee system for parameterized programs by reduction. The idea comes from [13], where the author proves that the rely-guarantee system is complete relative to the Owicki-Gries system, i.e. a rely-guarantee proof can be constructed from an Owicki-Gries proof of the same program. Thus, from the completeness of the Owicki-Gries system for parameterized programs it follows that the extended rely-guarantee system is also complete.

10 Example

We verify a simple parameterized parallel program presented in [13]. The program searches for the first element satisfying some predicate P in an array B of length m (represented by a list). If there is one, we call it *min-el*, otherwise *min-el* $= m$. Upon termination the program establishes the postcondition

$$\textit{min-el} < m{+}1 \wedge (\forall\, i{<}\textit{min-el}.\ \neg\ P(B!i)) \wedge \textit{min-el} < m \longrightarrow P(B!\textit{min-el}).$$

We use n concurrent programs, $S\ 0 \parallel \ldots \parallel S\ (n{-}1)$ (for simplicity let n divide m) such that $S\ i$ visits the array indices $i,\ n{+}i,\ 2{*}n{+}i, \ldots,\ m{+}i$.

Each $S\ i$ uses two variables x_i and y_i ranging over natural numbers. There exist several ways of formalizing program variables in such formalizations [9]. In the approach used here the state is represented by an Isabelle record type, whose fields are the program variables . For example, if a program has a boolean variable a and a variable b ranging over the natural numbers, the program state would be represented by the record:

record *Example* =
  a :: *bool*
  b :: *nat*

The advantages of this representation w.r.t. other approaches are discussed in [11]. In our example, parameterized variables are used. These can be implemented by lists or, more abstractly, functions from naturals into the corresponding value domain:

record *Parameterized-Example* =
  x :: *nat* $\Rightarrow$ *nat*
  y :: *nat* $\Rightarrow$ *nat*

Then, an indexed variable x_i in our program is represented by the function x applied to index i, i.e. $x\ i$. To distinguish program variables we write them in sans serif (e.g. x) in the following.

Each component program $S\ i$ is a while-program which uses the variable $\mathsf{x}\ i$, initially set to i, for searching. It terminates if

1. $P\ (B!(\mathsf{x}\ i))$ or,
2. $\mathsf{x}\ i > m$ or,
3. $S\ k$, with $k{\neq}i$, has found that $P\ (B!(\mathsf{x}\ k))$ for $\mathsf{x}\ k < \mathsf{x}\ i$.

Another variable $\mathsf{y}\ i$ is initially set to $m{+}i$ and, if $P(B!(\mathsf{x}\ i))$ holds, then $\mathsf{y}\ i$ is set to $\mathsf{x}\ i$. Then, the termination condition for $S\ i$ is: $\exists\, j{<}n.\ \mathsf{y}\ j \leq \mathsf{x}\ i$. The code of the parameterized parallel program is:

scheme $[0 \leq i < n]$
while $(\forall j < n.\ \mathsf{x}\ i < \mathsf{y}\ j)$ **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**

where $f\ (i := t)$ is Isabelle syntax for function update. Assignment to a parameterized variable is written $\mathsf{y} := \mathsf{y}\ (i := \mathsf{x}\ i)$ meaning that the variable $\mathsf{y}\ i$ is updated to $\mathsf{x}\ i$. The loop invariant is the predicate:

$$(\mathsf{x}\ i)\ \mathit{mod}\ n = i \wedge (\forall z < \mathsf{x}\ i.\ z\ \mathit{mod}\ n = i \longrightarrow \neg P\ (B\ !\ z)) \wedge \\ (\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ (\mathsf{y}\ i)) \wedge \mathsf{y}\ i \leq m{+}i).$$

Now we need to find *rely* and *guar* conditions for each component program. Observe that for each $S\ i$ the following holds:

1. The variables $\mathsf{x}\ i$ and $\mathsf{y}\ i$ are only changed by component i, thus the environment cannot affect their values or increase $\mathsf{y}\ j$ for $j{\neq}i$.
2. The program $S\ i$ cannot affect the variables $\mathsf{x}\ j$ and $\mathsf{y}\ j$ for $j{\neq}i$, and it does not increase the initial value of $\mathsf{y}\ i$.

We represent the value of a variable x after a transition by $\overline{\mathsf{x}}$. In the theorem below, we show each component i of the parallel composition as a tuple formed by the program code and its corresponding specification, i.e. the pre, rely, guar and postcondition. This theorem states that the full annotated program is derivable in the formalized rely-guarantee system:

theorem *Parameterized-Example*: $m\ \mathit{mod}\ n = 0 \Longrightarrow$
$\Vdash$ **cobegin**
scheme $[0 \leq i < n]$
(**while** $(\forall j < n.\ \mathsf{x}\ i < \mathsf{y}\ j)$ **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,
$\{\!|\ (\mathsf{x}\ i)\ \mathit{mod}\ n = i \wedge (\forall z < \mathsf{x}\ i.\ z\ \mathit{mod}\ n = i \longrightarrow \neg P\ (B\ !\ z)) \wedge$
 $(\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ (\mathsf{y}\ i)) \wedge \mathsf{y}\ i \leq m{+}i)\ |\!\}$,
$\{\!|\ (\forall j < n.\ i \neq j \longrightarrow \overline{\mathsf{y}}\ j \leq \mathsf{y}\ j) \wedge \mathsf{x}\ i = \overline{\mathsf{x}}\ i \wedge \mathsf{y}\ i = \overline{\mathsf{y}}\ i\ |\!\}$,
$\{\!|\ (\forall j < n.\ i \neq j \longrightarrow \mathsf{x}\ j = \overline{\mathsf{x}}\ j \wedge \mathsf{y}\ j = \overline{\mathsf{y}}\ j) \wedge \overline{\mathsf{y}}\ i \leq \mathsf{y}\ i\ |\!\}$,
$\{\!|\ (\mathsf{x}\ i)\ \mathit{mod}\ n = i \wedge (\forall z < \mathsf{x}\ i.\ z\ \mathit{mod}\ n = i \longrightarrow \neg P\ (B\ !\ z)) \wedge$
 $(\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ (\mathsf{y}\ i)) \wedge \mathsf{y}\ i \leq m{+}i) \wedge (\exists j < n.\ \mathsf{y}\ j \leq \mathsf{x}\ i)\ |\!\})$
coend
sat $[\{\!|\ \forall i < n.\ \mathsf{x}\ i = i \wedge \mathsf{y}\ i = m{+}i\ |\!\}, \{\!|\ \mathsf{x} = \overline{\mathsf{x}} \wedge \mathsf{y} = \overline{\mathsf{y}}\ |\!\}, \{\!|\ \mathit{True}\ |\!\},$
 $\{\!|\ \forall i < n.\ (\mathsf{x}\ i)\ \mathit{mod}\ n = i \wedge (\forall z < \mathsf{x}\ i.\ z\ \mathit{mod}\ n = i \longrightarrow \neg P\ (B\ !\ z)) \wedge$
 $(\mathsf{y}\ i < m \longrightarrow P\ (B\ !\ (\mathsf{y}\ i)) \wedge \mathsf{y}\ i \leq m{+}i) \wedge (\exists j < n.\ \mathsf{y}\ j \leq \mathsf{x}\ i)\ |\!\}]$

The expressions $\{\!|\ p\ |\!\}$ are concrete syntax for the set of states (or pairs of states) satisfying p. From the specifications of the components we establish the specification of the parallel program given by the four conditions after the keyword **sat**. The precondition of each component is the invariant shown above and the postcondition corresponds to the invariant and the negation of the loop guard.

The precondition $\{\!|\ \forall i{<}n.\ \mathsf{x}\ i = i \wedge \mathsf{y}\ i = m{+}i\ |\!\}$ of the parallel program gives the initial values of the variables. We consider the parallel program to be closed, meaning that the environment is empty. Thus, the corresponding *rely* condition simply states that the program variables are not modified by the environment. Analogously, the parallel program must not guarantee anything to the environment, so the guarantee condition is just *True*.

The proof is done by interactively applying the proof rules backwards until all the verification conditions are generated. It requires only one application of the consequence rule. The final verification conditions are proved with standard Isabelle tactics for simplification and natural deduction.

When all component programs terminate, the established postcondition implies that the element we were looking for, namely *min-el*, is the minimum of the set $\{y\ 0, \ldots, y\ (n-1)\}$. This is proven in the following lemma:

$$\begin{array}{l}
[\![\forall i{<}n.\ (x\ i)\ mod\ n{=}i \wedge (\forall z < x\ i.\ z\ mod\ n{=}i \longrightarrow \neg P\ (B\ !\ z)) \wedge \\
\quad (y\ i < m \longrightarrow P(B\ !\ (y\ i)) \wedge y\ i \le m{+}i) \wedge (\exists j < n.\ y\ j \le x\ i); \\
\quad min\text{-}el = minimum\ (map\ y\ [0..n{-}1])]\!] \Longrightarrow min\text{-}el < m{+}1 \wedge \\
\ (\forall i < min\text{-}el.\ \neg\ P\ (B\ !\ i)) \wedge min\text{-}el < m \longrightarrow P\ (B\ !\ min\text{-}el)
\end{array}$$

where the function *minimum* returns the least element of a list, in this case the list $[y\ 0, \ldots, y\ (n-1)]$. Hence, upon termination, the program establishes the postcondition announced at the beginning of the section for the indicated value of *min-el*.

11 Conclusions

We have presented the first formalization of the rely-guarantee system and its soundness proof in a theorem prover. This work represents another successful step towards the embedding of programming languages and their verification calculi in theorem provers. Another interesting contribution of this work is the extension of the formal treatment from the two-process to the n-process case, which is a technical challenge in formal verification, and whose soundness and completeness proofs have not been considered before.

The total number of specification lines for this formalization is 330. For the proof of soundness we proved 90 lemmas which were proven in 2200 lines (number of interactions with the theorem prover). Comparing it to the formalization of the Owicki-Gries system [8] with 220 lines of specification, 49 lemmas and 340 lines of proofs, it is clear that the rely-guarantee method is more involved. This is the price to obtain a compositional method: the underlying theory requires more work, but yields a simpler proof system.

Machine-checking soundness proofs is labour intensive, however, it produces not only highest confidence in the proof but also leads to optimizations and simpler formulations, which becomes crucial as languages are enriched with more complicated features. We hope that this work encourages further development in this area.

Acknowledgment I wish to thank Tobias Nipkow for helpful comments on a previous version of the paper and Jozef Hooman and Qiwen Xu for their interest and useful feedback about their own and other related work.

References

1. W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*, volume 54 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.
2. S. O. Ehmety and L. C. Paulson. Program composition in Isabelle/UNITY. In M. Charpentier and B. Sanders, editors, *Formal Methods for Parallel Programming: Theory and Application*, 2002.
3. C. Flanagan, S. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, volume 2305 of *LNCS*, pages 262–277. Springer-Verlag, 2002.
4. 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.
5. C. B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
6. K. L. McMillan. Circular compositional reasoning about liveness. In *Advances in Hardware Design and Verification Methods: IFIP WG10.5 (CHARME'99)*, volume 1703 of *LNCS*, pages 342–345. Springer-Verlag, 1999.
7. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
8. T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, pages 188–203. Springer-Verlag, 1999.
9. L. C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 23(6):1–30, 2001.
10. L. Prensa Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In M. Charpentier and B. Sanders, editors, *6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications. Held in conjunction with the 15th International Parallel and Distributed Processing Symposium*. IEEE CS Press, 2001.
11. L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002. Available at http://tumb1.biblio.tu-muenchen.de/publ/diss/allgemein.html.
12. J. Rushby. Formal verification of McMillan's compositional assume-guarantee rule. Technical report, Sept. 2001.
13. C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.
14. Q. Xu, W.-P. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. Technical Report 9502, Christian-Albrechts-Universität Kiel, March 1995.
15. 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.

This article was processed using the LaTeX macro package with LLNCS style