The Rely–Guarantee Method for Verifying Shared Variable Concurrent Programs*

Qiwen Xu[1], Willem-Paul de Roever[1] and Jifeng He[2]

[1]Institut für Informatik und Praktische Mathematik II, Christian-Albrechts-Universität, zu Kiel, Kiel, Germany
[2]Programming Research Group, Oxford University Computing Laboratory, Oxford, UK

Keywords: Specification and verification; Concurrency; Soundness and completeness; Partial and total correctness; Deadlock freedom; Compositionality; Rely–Guarantee formalism

Abstract. Compositional proof systems for shared variable concurrent programs can be devised by including the interference information in the specifications. The formalism falls into a category called *rely–guarantee* (or *assumption–commitment*), in which a specification is explicitly (syntactically) split into two corresponding parts. This paper summarises existing work on the rely-guarantee method and gives a systematic presentation. A proof system for partial correctness is given first, thereafter it is demonstrated how the relevant rules can be adapted to verify deadlock freedom and convergence. Soundness and completeness, of which the completeness proof is new, are studied with respect to an operational model. We observe that the rely–guarantee method is in a sense a reformulation of the classical non-compositional Owicki & Gries method, and we discuss throughout the paper the connection between these two methods.

1. Introduction

In traditional verification methods for shared variable concurrent programs such as the well-known Owicki & Gries system [OwG76], verification can only be performed after the whole system has been constructed. This is considered unsatisfactory from the development point of view for two reasons. First, the full

---

implementation usually is very complicated and takes a long time to build. It is therefore in general very difficult to verify the system when the whole construction has been finished, because by then all the useful intuition used during the design process has been lost, and the verification can be as hard as building the whole system from the scratch again. The second and probably more compelling reason is that if the verification is only conducted after all the implementations have been completed, an early design error will cause most of the later design work to be discarded. Therefore in a constructive methodology, a specification is moved gradually, step by step, to its final implementation. Each design step is checked as soon as it is carried out, and the correctness of these individual steps together guarantees the correctness of the whole system, so that when one design step goes wrong, the designers only need to return to the place before that step and start from there, instead of the very beginning again.

Therefore, there has been a considerable amount of effort recently in developing what are known as compositional methods; they allow the satisfaction of a specification by a system be verified on the basis of specifications of its constituent components, without knowing the interior construction of those components. The major obstacle to compositionality in concurrency is the possible interference among processes. For example, to cope with interference, Owicki and Gries introduced an interference freedom test, which ensures that no assertions used in the proof of one process are invalidated by the execution of another process. Their interference freedom test was formulated in such a way that the knowledge of the complete system code is assumed. The key point to achieve compositionality is to reformulate this interference freedom test.

Actually, what is really relevant is not the structure of the interfering agents, but the effect of the interference. In a shared variable setting, the interference between a component and its environment occurs when one or both of them updates a common state. Jones [Jon81] proposed to include such information in the specification, describing by a *rely*-condition and a *guar*-condition state changes from an environment and a component respectively. In this way, compositionality can indeed be achieved, and an outline of a compositional proof system for a large subset of the Owicki & Gries language was given in [Jon81]. An initial attempt to give a semantic model for Jones' system was made by Aczel in an unpublished note, and the results were cited in [deRo85]. The first mathematical treatment of this approach (for a similar system for the full Owicki & Gries language) was developed by Stirling [Sti88]. Stirling proved that his system is sound and also indicated that his system is complete relative to the Owicki & Gries method. Stirling only dealt with partial correctness, but he provided an elegant framework, after which a number of more recent works have followed, e.g., [Stø90, Stø91a, Stø91b, XuH91, Xu92], addressing also deadlock freedom and convergence.

A large part of this paper is a synthesis of existing work of several authors, in particular, Jones, Stirling, Stølen and ourselves. The first contribution of this paper is a systematic presentation: to begin with, a system for partial correctness is presented, thereafter it is modified for verifying deadlock freedom and convergence. The second contribution is a thorough discussion of the theoretical foundation, especially the completeness issue. The last contribution is our observation that the rely–guarantee method is a reformulation of the classical Owicki & Gries method, and revealing the connection between these two methods offers a deep understanding about verification of shared variable programs.

The paper is structured as follows: Section 2 is a brief introduction to a programming language and an operational model of it. A proof system for

partial correctness is given in Section 3. In Sections 4 and 5, we discuss how the rules can be modified to verify deadlock freedom and convergence. In Sections 3 and 4, soundness and completeness are discussed for the concerned proof systems. A short discussion in Section 6 about related topics concludes the paper.

2. The Language

2.1. Syntax

The language used in this paper is adopted from [OwG76]. It basically extends Dijkstra's guarded command language by parallel composition and a simple synchronisation statement. The syntax and a brief explanation of various language structures are as follows:

$$P ::= \bar{x} := \bar{e} \mid P_1;P_2 \mid \textbf{if } b_1 \rightarrow P_1 \square \cdots \square b_n \rightarrow P_n \textbf{ fi} \mid \textbf{while } b \textbf{ do } P \textbf{ od} \mid$$
$$\textbf{await } b \textbf{ then } P \textbf{ end} \mid P_1 \parallel P_2$$

In the assignment statement, $\bar{x}$ represents a vector of variables $(x_1, \ldots, x_n)$, and $\bar{e}$ represents a vector of expressions $(e_1, \ldots, e_n)$; both of them are of the same length. The computations of expressions $e_1, \ldots, e_n$ in the assignment statement are assumed to be terminating, and moreover the execution the complete statement is considered as atomic, with $x_1, \ldots, x_n$ being set to the values computed for $e_1, \ldots, e_n$. A vacuous assignment, such as $\bar{x} := \bar{x}$, is shortened to **skip**. In a sequential composition $P;Q$, P is executed first, and if it terminates, Q is then executed. When more than one guards holds in the conditional statement **if** $b_1 \rightarrow P_1 \square \cdots \square b_n \rightarrow P_n$ **fi**, any of the corresponding programs can be selected for execution; when none of the guards hold, the process terminates. Iteration is represented by **while** b **do** P **od** as usual. In both conditional and iteration statements, the evaluation of the boolean tests are atomic, but an environment can interrupt between the boolean tests and the first action from the corresponding program bodies.

The two more complicated structures in this language are the parallel composition and the await statement. The execution of each process is composed of atomic actions, and concurrency is modelled by nondeterministic interleaving of these actions of the constituent processes. In $P \parallel Q$, P and Q are executed concurrently, with atomic actions from two processes interleaving each other. Process $P \parallel Q$ is blocked if and only if both subprocesses are blocked. Parallel composition of n processes $P_1 \parallel (P_2 \parallel (\ldots \parallel P_n)\ldots)$, is simply written as $P_1 \parallel P_2 \parallel \ldots \parallel P_n$. Synchronisation and mutual exclusion are achieved by **await** b **then** P **end**, which in turn gives rise to the possibility of deadlock. When b holds, P will be executed without interruption; when b does not hold, the process is blocked and can only become active when the environment has set b to true. The meaning of an await statement is not very clear when its body does not terminate. For partial correctness, this is not a problem, because one is only required to show that any terminated state satisfies the post-condition. However, for total correctness, such behaviours should be ruled out. For the uniformity of the paper, we stipulate that an await body always terminates, and we ensure this by disallowing iteration and await structures in the body.

2.2. Operational Semantics

A state is a mapping from program variables to some values. A configuration is a pair $\langle P, \eta \rangle$, where P is either a program, or a special symbol E standing for the end of a program, and η is a state. A transition is represented by a labelled arrow connecting the beginning and ending configurations. It is either of the form $\langle P, \eta \rangle \xrightarrow{c} \langle P', \eta' \rangle$, for a step from the component, or of the form $\langle P, \eta \rangle \xrightarrow{e} \langle P, \eta' \rangle$, for a step from the environment. That is, component and environment transitions are labelled by c and e respectively. Note also that the program part is not changed in an environment transition. A component transition is enabled at a configuration $\langle P, \eta \rangle$, denoted by $\langle P, \eta \rangle \xrightarrow{c}$, if there exist P' and η' such that $\langle P, \eta \rangle \xrightarrow{c} \langle P', \eta' \rangle$; otherwise it is not enabled, and denoted by $\neg \langle P, \eta \rangle \xrightarrow{c}$. An environment transition is always enabled. The idea of dividing the actions into component and environment ones was first suggested by Aczel. The important contribution of it is that this leads to a compositional semantics, namely, the semantics of a compound statement can be defined as a function of the semantics of the two constituent statements. For the simplicity of the presentation, we define the semantics along the line of [Sti88] in the style of Plotkin's Structured Operational Semantics [Plo81], and show later that the semantics is indeed compositional.

The rule for an environment transition is:

$$\langle P, \eta \rangle \xrightarrow{e} \langle P, \eta' \rangle, \text{ for arbitrary states } \eta \text{ and } \eta'$$

This again reflects the compositionality of the semantics: to model possible transitions from the environment, any state changes are allowed. Later, when processes are composed, the environment transitions of one process would be component transitions of another process, and the transitions whose occurrence is not possible will be ruled out.

The rules for component transitions are as follows:

$\langle \bar{x} := \bar{e}, \eta \rangle \xrightarrow{c} \langle E, (\eta : \bar{x} \mapsto \bar{e}) \rangle$
>   Here $(\eta : \bar{x} \mapsto \bar{e})$ stands for a state function which is same as η
>   except mapping $\bar{x}$ to $\bar{e}(\eta)$ ($\bar{e}(\eta)$ is the value of expression $\bar{e}$ in state η)

$\langle P ; Q, \eta \rangle \xrightarrow{c} \langle P' ; Q, \eta' \rangle$, if $\langle P, \eta \rangle \xrightarrow{c} \langle P', \eta' \rangle$

$\langle \mathbf{if}\, b_1 \rightarrow P_1 \square \cdots \square b_n \rightarrow P_n\, \mathbf{fi},\, \eta \rangle \xrightarrow{c} \langle P_i, \eta \rangle$, if $\eta \models b_i$ holds
$\langle \mathbf{if}\, b_1 \rightarrow P_1 \square \cdots \square b_n \rightarrow P_n\, \mathbf{fi},\, \eta \rangle \xrightarrow{c} \langle E, \eta \rangle$, if $\eta \not\models (b_1 \vee \cdots \vee b_n)$ holds

$\langle \mathbf{while}\, b\, \mathbf{do}\, P\, \mathbf{od},\, \eta \rangle \xrightarrow{c} \langle P ; \mathbf{while}\, b\, \mathbf{do}\, P\, \mathbf{od},\, \eta \rangle$, if $\eta \models b$ holds
$\langle \mathbf{while}\, b\, \mathbf{do}\, P\, \mathbf{od},\, \eta \rangle \xrightarrow{c} \langle E, \eta \rangle$, if $\eta \not\models b$ holds

$\langle \mathbf{await}\, b\, \mathbf{then}\, P\, \mathbf{end},\, \eta \rangle \xrightarrow{c} \langle E, \eta' \rangle$, if $\eta \models b$ holds and there exist $P_1, \ldots, P_n$,
> $\eta_1, \ldots, \eta_n$ such that $\langle P, \eta \rangle \xrightarrow{c} \langle P_1, \eta_1 \rangle \xrightarrow{c} \cdots \xrightarrow{c} \langle P_n, \eta_n \rangle \xrightarrow{c} \langle E, \eta' \rangle$

$\langle P \parallel Q, \eta \rangle \xrightarrow{c} \langle P' \parallel Q, \eta' \rangle$, if $\langle P, \eta \rangle \xrightarrow{c} \langle P', \eta' \rangle$
$\langle P \parallel Q, \eta \rangle \xrightarrow{c} \langle P \parallel Q', \eta' \rangle$, if $\langle Q, \eta \rangle \xrightarrow{c} \langle Q', \eta' \rangle$

For notational convenience, let $E;P \stackrel{\text{def}}{=} P$, $P \parallel E \stackrel{\text{def}}{=} P$ and $E \parallel P \stackrel{\text{def}}{=} P$. This allows the following transition rules

$\langle P;Q,\eta \rangle \stackrel{c}{\rightarrow} \langle Q,\eta' \rangle$, if $\langle P,\eta \rangle \stackrel{c}{\rightarrow} \langle E,\eta' \rangle$

$\langle P \parallel Q,\eta \rangle \stackrel{c}{\rightarrow} \langle Q,\eta' \rangle$, if $\langle P,\eta \rangle \stackrel{c}{\rightarrow} \langle E,\eta' \rangle$

$\langle P \parallel Q,\eta \rangle \stackrel{c}{\rightarrow} \langle P,\eta' \rangle$, if $\langle Q,\eta \rangle \stackrel{c}{\rightarrow} \langle E,\eta' \rangle$

to be included as special cases of the ones presented before.

2.3. Computation

For a program P, a computation is any finite or infinite sequence

$$\langle P_0,\eta_0 \rangle \stackrel{\delta_1}{\rightarrow} \langle P_1,\eta_1 \rangle \stackrel{\delta_2}{\rightarrow} \cdots \stackrel{\delta_n}{\rightarrow} \langle P_n,\eta_n \rangle \stackrel{\delta_{n+1}}{\rightarrow} \cdots, \quad \delta_i \in \{e,c\}, \quad i = 1,\ldots,n\ldots$$

where each transition satisfies the transition relation and $P_0 = P$. We write $cp[P]$ for the set of all computations of P. Obviously, by this definition, $cp[P]$ is prefix closed. If σ denotes the above computation, then define

$\sigma_i \stackrel{\text{def}}{=} \langle P_i,\eta_i \rangle$ the (i+1)-th configuration

$Tr(\sigma,i) \stackrel{\text{def}}{=} \delta_i$ the i-th transition label

$Pr(\sigma_i) \stackrel{\text{def}}{=} P_i$ the (i+1)-th program component

$St(\sigma_i) \stackrel{\text{def}}{=} \eta_i$ the (i+1)-th state component

Moreover, $\sigma[i,\ldots,j]$ represents the segment of σ which starts with configuration σ_i and ends with configuration σ_j. If σ is finite, its length $len(\sigma)$ is the number of transitions in it and σ_{last} is $\sigma_{len(\sigma)}$; if σ is infinite, $len(\sigma)$ is ∞.

3. A Proof System for Partial Correctness

In this section, we consider verification of partial correctness, where the main focus is to establish a post-condition for final states of terminated computations. The rely–guarantee method was first proposed over ten years ago, but its close connection to Owicki & Gries method only becomes obvious in our recent search for a completeness proof. Let us consider an example:

$P_1 :: x := x + 1 \parallel P_2 :: x := x + 2$

where P_1 and P_2 are names of the processes. The execution of an assignment statement is considered atomic, so if started with the value of x equal to 0, this program ends with value of x equal to 3. In Owicki & Gries method, the first step is to annotate an assertion to every control point of each process and show that the processes are locally correct (as if they were run in isolation) w.r.t these assertions. For our example, we choose p_1, q_1, p_2 and q_2 as $x = 0 \vee x = 2$, $x = 1 \vee x = 3$, $x = 0 \vee x = 1$ and $x = 2 \vee x = 3$ respectively, and local verification consists of :

$\{p_1\}\ x := x + 1\ \{q_1\}$
$\{p_2\}\ x := x + 2\ \{q_2\}$

Next, the interference freedom test is performed, in which every assertion used in the local verification is shown not invalidated by the execution of the other process. In this simple case, it consists of four simple proofs:

$$\{p_1 \wedge p_2\} \; x := x + 2 \; \{p_1\}$$
$$\{q_1 \wedge p_2\} \; x := x + 2 \; \{q_1\}$$
$$\{p_1 \wedge p_2\} \; x := x + 1 \; \{p_2\}$$
$$\{p_1 \wedge q_2\} \; x := x + 1 \; \{q_2\}$$

Although this interference freedom test is formulated with the complete code of the system, one could look at it from a different angle. In the example, the transition of P_2 (which is considered as the environment of P_1) is constrained by the predicate $(x = 0 \vee x = 1) \wedge (x' = 2 \vee x' = 3)$, where x and x' refer to program states before and after a transition. This fact suffices to ensure that the assertions used in P_1's local proof are not invalidated. Conversely, the same holds for P_2. This indicates that when interference information is recorded in a specification, it can be used in the verification of constituent processes, and consequently no additional interference freedom test is needed, thus making the method compositional.

3.1. Specification and Correctness

A specification describes conditions (called *assumptions*) under which the program is used, and the expected behaviours (called *commitments*) of the program when it is used under these conditions. The specification proposed by Jones [Jon81] consists of a quadruple of predicates:

(*pre, rely, guar, post*)

where the assumption is composed of *pre* and *rely*, and the commitment is composed of *guar* and *post*. Besides program variables, *logical* (sometimes also called *freeze* or *rigid*) variables are used in specifications. Logical variables do not occur in programs, hence, their values are not changed during the execution and consequently not recorded in states. In this paper, we use y to denote the vector of program variables. Logical variables are usually indexed by 0 to distinguish them from program variables (this also means program variables should not use symbols indexed by 0), and denote the vector of all the logical variables by y_0. Let y' be y with each of its elements primed; it is used to refer to program states after a transition. The parameter dependence of the specification predicates is given by $pre(y, y_0)$, $rely(y, y', y_0)$, $guar(y, y', y_0)$ and $post(y, y_0)$.

Informally, a program P satisfies such a specification, denoted by the correctness formula $P \; \underline{sat} \; (pre, rely, guar, post)$, if

1) P is invoked in a state which satisfies *pre*, and

2) any environment transition satisfies *rely*,

then

3) any component transition satisfies *guar*,

4) if a computation terminates, the final state satisfies *post*.

Jones originally suggested that *rely* and *guar* should be both reflexive and transitive (w.r.t. their first two parameters). This reflects the abstraction of the semantics at the proof theoretic level. However, specifications become difficult to write when only reflexive and transitive predicates are allowed as rely- and guarantee-conditions. Therefore, in favour of concise formulation, we only require these predicates to be reflexive. This position is similar to that of Lamport in his

Temporal Logic of Actions [Lam95], where the so-called stuttering transitions are built in the semantics.

We now formalise these notions. In the following, let γ be an arbitrary valuation for logical variables. Clearly, for a configuration Γ, only the state part of it gives valuation for program variables, but we shall also write simply $(\Gamma, \gamma) \models pre$, etc., since this causes no confusion. Let σ be a computation, and

$$A(pre, rely)(\gamma) \stackrel{\text{def}}{=} \{\sigma \mid (\sigma_o, \gamma) \models pre, \text{ and for any transition}$$
$$\sigma_i \stackrel{e}{\to} \sigma_{i+1} \text{ in } \sigma,\ (\sigma_i, \sigma_{i+1}, \gamma) \models rely\}$$
$$C(guar, post)(\gamma) \stackrel{\text{def}}{=} \{\sigma \mid \text{for any transition } \sigma_i \stackrel{c}{\to} \sigma_{i+1} \text{ in } \sigma,$$
$$(\sigma_i, \sigma_{i+1}, \gamma) \models guar,$$
$$\text{and if } len(\sigma) < \infty \land Pr(\sigma_{last}) = E,$$
$$\text{then } (\sigma_{last}, \gamma) \models post\}$$

Validity of the correctness formula is defined by

$$\models P \ \underline{sat} \ (pre, rely, guar, post)$$
$$\stackrel{\text{def}}{=} \forall \gamma. cp[P] \cap A(pre, rely)(\gamma) \subseteq C(guar, post)(\gamma)$$

Valid computations are those such that if they satisfy assumptions, then they also satisfy commitments. A program satisfies a specification if all its computations are valid. Note that in a computation, environment transitions are allowed even when the component has completed execution (that is, has reached a configuration with an empty program), and therefore the post-condition must be satisfied not only in the states in which the component has finished the last transition, but also in the following states after any number of environment transitions.

Although computations can be both finite and infinite, we only need to check the finite computations in verifying partial correctness, because for any program if there is an infinite computation which is invalid, there is also an invalid finite computation. Therefore, if all the finite computations are valid, then all the infinite computations are valid too. The same is true in the next section where we discuss verification of deadlock freedom. As a matter of fact, if we were only interested in verifying partial correctness and deadlock freedom, we could restrict $cp[P]$ to finite computations only. However, this is not sufficient when we verify convergence.

3.2. Proof System

In this section, we present the proof system together with some toy examples of their applications. First we introduce some notations. For two predicates $f(y, y_0)$ and $g(y, y', y_0)$, let f $\underline{stable\ when}$ g be a shorthand for $\forall y, y', y_0. f(y, y_0) \land g(y, y', y_0) \to f(y', y_0)$ and $f'(y, y_0)$ denote $f(y', y_0)$.

The proof system has two axioms concerning the two atomic statements assignment and await.

Assignment axiom

$$\frac{\begin{array}{l} pre \to post[\bar{e}/\bar{x}] \\ (pre \land \lceil \bar{x}' = \bar{e} \rceil) \to guar \\ pre \ \underline{stable\ when}\ rely \\ post \ \underline{stable\ when}\ rely \end{array}}{\bar{x} := \bar{e} \ \underline{sat} \ (pre, rely, guar, post)}$$

where $\lceil \bar{x}' = \bar{e} \rceil \stackrel{\text{def}}{=} (\bar{x}' = \bar{e} \vee x' = x) \wedge \forall z \in (y - \bar{x}). z' = z$. There is exactly one component transition in the assignment, which assigns $\bar{e}$ to $\bar{x}$ and leaves other state variables unchanged; it therefore satisfies $\lceil \bar{x}' = \bar{e} \rceil$. In a typical computation, there are a number of environment transitions before and after the component transition. Since *pre* holds initially, it follows from *pre stable when rely* that *pre* still holds immediately before the component transition. From *pre* → *post*[$\bar{e}/\bar{x}$], it follows that *post* holds immediately after the component transition, hence, due to *post stable when rely*, *post* holds in any states after a number of environment transitions. An example of an application of this rule is:

$$x := 10 \ \underline{sat} \ (true, \ x > 0 \rightarrow x' \geqslant x, \ true, \ x \geqslant 10)$$

Await axiom

$$\frac{\begin{array}{l} pre \ \underline{stable \ when} \ rely \\ post \ \underline{stable \ when} \ rely \\ P \ \ \underline{sat} \ \ (pre \wedge b \wedge y = v_0, \ y' = y, \ true, \ guar[v_0/y, \ y/y'] \wedge post) \end{array}}{\textbf{await} \ b \ \textbf{then} \ P \ \textbf{end} \ \ \underline{sat} \ \ (pre, \ rely, \ guar, \ post)}$$

where v_0 is a fresh vector of logical variables. The await statement says that its body is executed when b holds and is atomic. Therefore the state transition caused by P (that is, the complete input/output mapping by P, and not the individual transitions in P) should satisfy *guar*. Since the guarantee-condition describes state transitions while the post-condition describes states only, v_0 is used to record the state before the execution of P, so that the verification of the guarantee-condition for the await statement is transformed to the verification of a post-condition for the body P. Await statement

$$\textbf{await} \ x > 0 \ \textbf{then} \ x := x - 1 \ \textbf{end}$$

satisfies

$$(x \geqslant 0, \ x \geqslant 0 \rightarrow x' \geqslant 0, \ x' \leqslant x, \ x \geqslant 0)$$

Consequence rule

$$\frac{\begin{array}{l} pre \rightarrow pre_1, \ rely \rightarrow rely_1, \ guar_1 \rightarrow guar, \ post_1 \rightarrow post \\ P \ \ \underline{sat} \ \ (pre_1, \ rely_1, \ guar_1, \ post_1) \end{array}}{P \ \ \underline{sat} \ \ (pre, \ rely, \ guar, \ post)}$$

This rule allows one to strengthen the assumptions and weaken the commitments of a specification. Consider the example used to illustrate the assignment axiom. By applying the consequence rule, we obtain

$$x := 10 \ \underline{sat} \ (x = -2, \ x > 0 \rightarrow x' \geqslant x, \ true, \ x \geqslant 10 \vee x = -6)$$

Sequential composition rule

$$\frac{\begin{array}{l} P \ \ \underline{sat} \ \ (pre, \ rely, \ guar, \ mid) \\ Q \ \ \underline{sat} \ \ (mid, \ rely, \ guar, \ post) \end{array}}{P \ ; Q \ \ \underline{sat} \ \ (pre, \ rely, \ guar, \ post)}$$

As an example, from

$$x := x + 1 \ \underline{sat} \ (x \geqslant x_0, \ x_0 \leqslant x \rightarrow x \leqslant x', \ x' \geqslant x, \ x \geqslant x_0 + 1)$$
$$x := x + 1 \ \underline{sat} \ (x \geqslant x_0 + 1, \ x_0 \leqslant x \rightarrow x \leqslant x', \ x' \geqslant x, \ x \geqslant x_0 + 2)$$

where x_0 is a logical variable, by the sequential composition rule, we have

$x := x + 1 \, ; \, x := x + 1 \, \underline{sat} \, (x \geqslant x_0, \, x_0 \leqslant x \to x \leqslant x', \, x' \geqslant x, \, x \geqslant x_0 + 2)$

Conditional rule

$$\frac{\begin{array}{l} pre \, \underline{stable \, when} \, rely \\ P_i \, \underline{sat} \, (pre \wedge b_i, \, rely, \, guar, \, post) \\ \textbf{skip} \, \underline{sat} \, (pre \wedge \neg(b_1 \vee \ldots \vee b_n)), \, rely, \, guar, \, post) \end{array}}{\textbf{if} \, b_1 \to P_1 \square \cdots \square b_n \to P_n \, \textbf{fi} \, \underline{sat} \, (pre, \, rely, \, guar, \, post)}$$

The more common conditional statement **if** b **then** P **else** Q **fi** can be defined as **if** $b \to P \square \neg b \to Q$ **fi**. By our convention, when none of the guards hold, the statement terminates, and therefore **if** $b \to P$ **fi** is the same as **if** b **then** P **else skip fi**. Once we have shown

$x := 10 \, \underline{sat} \, (x < 10, \, x \leqslant x', \, x \leqslant x', \, x \geqslant 10)$ and
skip $\underline{sat} \, (x \geqslant 10, \, x \leqslant x', \, x \leqslant x', \, x \geqslant 10)$

we can apply this rule to obtain,

if $x < 10 \to x := 10$ **fi** $\underline{sat} \, (true, \, x \leqslant x', \, x \leqslant x', \, x \geqslant 10)$

The additional condition *pre* *stable when* *rely* is necessary, since the pre-condition is only required to hold initially in the interpretation of a correctness formula. Without this assumption, *pre* may be violated by environment transitions, consequently causing the hypotheses P_i $\underline{sat}$ $(pre \wedge b_i, \, rely, \, guar, \, post)$ and **skip** $\underline{sat}$ $(pre \wedge \neg(b_1 \vee \ldots \vee b_n)), \, rely, \, guar, \, post)$ to become inapplicable.

Iteration rule

$$\frac{\begin{array}{l} pre \, \underline{stable \, when} \, rely \\ pre \wedge \neg b \to post \\ post \, \underline{stable \, when} \, rely \\ P \, \underline{sat} \, (pre \wedge b, \, rely, \, guar, \, pre) \end{array}}{\textbf{while} \, b \, \textbf{do} \, P \, \textbf{od} \, \underline{sat} \, (pre, \, rely, \, guar, \, post)}$$

In this rule, assertion *pre* is an invariant for the loop. Note that it does not have to hold in the middle of the execution of the loop body; what is needed is that the invariant holds initially and after each iteration. When we have shown

$x := x + 1 \, \underline{sat} \, (x \leqslant 10, \, x \leqslant x', \, x \leqslant x', \, true)$

by using the rule, we can deduce

while $x \leqslant 10$ **do** $x := x + 1$ **od** $\underline{sat} \, (true, \, x \leqslant x', \, x \leqslant x', \, x > 10)$.

Parallel rule

$$\frac{\begin{array}{l} (rely \vee guar_1) \to rely_2 \\ (rely \vee guar_2) \to rely_1 \\ (guar_1 \vee guar_2) \to guar \\ P \, \underline{sat} \, (pre, \, rely_1, \, guar_1, \, post_1) \\ Q \, \underline{sat} \, (pre, \, rely_2, \, guar_2, \, post_2) \end{array}}{P \parallel Q \, \underline{sat} \, (pre, \, rely, \, guar, \, post_1 \wedge post_2)}$$

Here the program of interest is $P \parallel Q$. Assume the overall environment is R. The environment of process P consists of Q and R, and the environment of process Q consists of P and R. Therefore, process P should be able to tolerate interference

from both R and Q. Hence, the strongest rely-condition that P can assume is $rely \vee guar_2$. For the same reason, the strongest rely-condition for Q is $rely \vee guar_1$. As a more concrete example, assume the overall environment does not increase the value of x, captured by the rely-condition $x' \leqslant x$, and the transitions from P will not decrease the value of x, captured by $guar_1$ defined as $x' \geqslant x$. However, process Q in this case must be able to behave properly under the interference which may both increase and decrease the value of x.

It is easy to see that a component transition of $P \parallel Q$ is either from P or from Q, and hence it satisfies $guar_1 \vee guar_2$. When both P and Q have terminated, from the two sub-specifications it follows that both $post_1$ and $post_2$ are satisfied.

Just as in the Owicki & Gries method, auxiliary variables are needed in some cases to characterise the intermediate states of the executions. The following rule allows properties of a program to be deduced from properties of the program augmented with auxiliary variables.

Auxiliary variable rule

$$\frac{\exists z.pre_1(y, z, y_0) \\ \exists z'.rely_1((y, z), (y', z'), y_0) \\ P \ \underline{sat} \ (pre \wedge pre_1, \ rely \wedge rely_1, \ guar, \ post)}{Q \ \underline{sat} \ (pre, \ rely, \ guar, \ post)}$$

provided

    i. z is a list of auxiliary variables in P, and any member of z appears either in the left-hand side of an assignment, or in the right-hand side but only if the corresponding variable in the left-hand side is also one in z;

    ii. P coincides with Q when z is removed from the former;

    iii. $z \cap freevar(pre, rely, guar, post) = \emptyset$, where $freevar(pre, rely, guar, post)$ is the set of free variables in the specification.

In this rule, pre, $rely$, $guar$ and $post$ do not contain free occurrences of any variables in z, while pre_1, $rely_1$ do not constrain the program variables in the sense that $\exists z.pre_1(y, z, y_0)$ and $\exists z'.rely_1((y, z), (y', z'), y_0)$ are valid.

Without this rule, the proof system in this paper would be incomplete. That is, some valid correctness formulas would not be deducible in the proof system. We do not restrict the kinds of auxiliary variables. However, later, in the completeness proof, only a special kind of auxiliary variable (namely a history variable) is needed. Nevertheless, experiments have shown that it is easier to write specifications in which auxiliary variables are not restricted to such a special kind.

We illustrate the use of the last two rules by an example.

$$x := x + 1 \parallel x := x + 1$$

This simple program has been used many times to test whether a proof system for parallel programs is compositional. Let $P1$ be the program $(x, z_1 := x + 1, z_1 + 1)$, and $P2$ the program $(x, z_2 := x + 1, z_2 + 1)$, then $P1$ satisfies

$$\begin{aligned} pre &: \quad x = z_1 = z_2 = 0 \\ rely &: \quad (x = z_1 + z_2 \rightarrow x' = z_1' + z_2') \wedge z_1' = z_1 \\ guar &: \quad (x = z_1 + z_2 \rightarrow x' = z_1' + z_2') \wedge z_2' = z_2 \\ post &: \quad x = z_1 + z_2 \wedge z_1 = 1 \end{aligned}$$

and $P2$ satisfies

pre : $x = z_1 = z_2 = 0$
$rely$: $(x = z_1 + z_2 \rightarrow x' = z_1' + z_2') \wedge z_2' = z_2$
$guar$: $(x = z_1 + z_2 \rightarrow x' = z_1' + z_2') \wedge z_1' = z_1$
$post$: $x = z_1 + z_2 \wedge z_2 = 1$

By the parallel composition and consequence rules, $P1 \parallel P2$ satisfies

pre : $x = z_1 = z_2 = 0$
$rely$: $x' = x \wedge z_1' = z_1 \wedge z_2' = z_2$
$guar$: $true$
$post$: $x = 2$

Finally by the auxiliary variable rule, we get

$$x := x + 1 \parallel x := x + 1 \quad \underline{sat} \quad (x = 0,\ x' = x,\ true,\ x = 2)$$

3.3. Soundness and Completeness

This section is devoted to soundness and completeness of the system. Throughout the paper, we assume an arbitrary language in which the assertions are expressed. As a usual practice when concentrating on a program logic, we assume that the logic for the underlying data domain is sound and complete, namely,
    $\vdash r$ iff $\models r$, for every assertion r
Completeness results of this sort are known as relative completeness [Coo78].

The soundness theorem says that if one deduces that a program P satisfies a specification (pre, $rely$, $guar$, $post$) in the system, that is, $\vdash P$ $\underline{sat}$ (pre, $rely$, $guar$, $post$), then this is true also semantically, namely $\models P$ $\underline{sat}$ (pre, $rely$, $guar$, $post$).

Theorem 1. (Soundness) The proof system in this section is sound.

In this paper, we concentrate on parallel composition. Detailed proofs concerning other structures can be found in [XdR95]. First, we show how to decompose a computation in a parallel composition into computations of its subprocesses.

Definition 1. Computations $\sigma \in cp[P \parallel Q]$, $\sigma^1 \in cp[P]$ and $\sigma^2 \in cp[Q]$ *conjoin*, denoted by $\sigma \propto \sigma^1 \parallel \sigma^2$, if and only if

i. $len(\sigma) = len(\sigma^1) = len(\sigma^2)$
ii. $St(\sigma_i) = St(\sigma_i^1) = St(\sigma_i^2)$,   for $0 \leqslant i \leqslant len(\sigma)$
iii. for $1 \leqslant i \leqslant len(\sigma)$, one of the following three cases holds :
        $Tr(\sigma^1, i) = c, Tr(\sigma^2, i) = e$ and $Tr(\sigma, i) = c$
        $Tr(\sigma^1, i) = e, Tr(\sigma^2, i) = c$ and $Tr(\sigma, i) = c$
        $Tr(\sigma^1, i) = e, Tr(\sigma^2, i) = e$ and $Tr(\sigma, i) = e$
iv. $Pr(\sigma_i) = Pr(\sigma_i^1) \parallel Pr(\sigma_i^2)$,   for $0 \leqslant i \leqslant len(\sigma)$

Two computations of P and Q can be combined into a computation of $P \parallel Q$, if and only if they have the same state sequence and do not have c-steps at the same time. The resulting computation of $P \parallel Q$ also has the same state sequence. Furthermore, in this computation a transition is labelled as c if this is the label in one of the computations of P and Q at the corresponding position; a transition is labelled as e if this is the case in both computations of P and Q at the corresponding position.

Lemma 1.

$cp[P \parallel Q] = \{\sigma \mid \text{there exist } \sigma^1 \in cp[P] \text{ and } \sigma^2 \in cp[Q], \text{ such that } \sigma \propto \sigma^1 \parallel \sigma^2\}$

Proof. Direct from the transition rules. $\Box$

This lemma indicates that the semantics is compositional.

Soundness of the parallel composition rule. Let γ be an arbitrary logical valuation and assume that the following holds:

$$\models P \ \underline{sat} \ (pre, \ rely_1, \ guar_1, \ post_1)$$
$$\models Q \ \underline{sat} \ (pre, \ rely_2, \ guar_2, \ post_2)$$
$$\models (rely \vee guar_1) \rightarrow rely_2$$
$$\models (rely \vee guar_2) \rightarrow rely_1$$
$$\models (guar_1 \vee guar_2) \rightarrow guar$$

Lemma 2. If $\sigma \in A(pre, rely)(\gamma) \cap cp[P \parallel Q]$ and $\sigma \propto \sigma^1 \parallel \sigma^2$ with $\sigma^1 \in cp[P]$ and $\sigma^2 \in cp[Q]$, then each c-transition in σ^1 and σ^2 satisfies $guar_1$ or $guar_2$, respectively.

Proof. If this is not the case, assume that the first c-transition which does not satisfy the guarantee-condition is from P at step k. From lemma 1, each e-transition in $\sigma^1[0 \ldots k]$ corresponds to a c-transition in σ^2 or a e-transition in σ, therefore it satisfies $rely \vee guar_2$. Hence, $\sigma^1[0 \ldots k] \in A(pre, \ rely_1)(\gamma)$, but this contradicts $\models P \ \underline{sat} \ (pre, \ rely_1, \ guar_1, \ post_1)$, because one c-transition in $\sigma^1[0 \ldots k]$ does not satisfy the guarantee-condition. $\Box$

Lemma 3. If $\sigma \in A(pre, rely)(\gamma) \cap cp[P \parallel Q]$ and $\sigma \propto \sigma^1 \parallel \sigma^2$ with $\sigma^1 \in cp[P]$ and $\sigma^2 \in cp[Q]$, then each e-transition in σ^1 and σ^2 satisfies $rely \vee guar_2$ or $rely \vee guar_1$ respectively.

Proof. From lemmas 1 and 2. $\Box$

Lemma 4. If $\sigma \in A(pre, rely)(\gamma) \cap cp[P \parallel Q]$, then each c-transition in σ satisfies $guar$.

Proof. From lemma 2. $\Box$

Lemma 5. If $\sigma \in A(pre, rely)(\gamma) \cap cp[P \parallel Q]$, $len(\sigma) < \infty$, and $Pr(\sigma_{last}) = E$, then $(\sigma_{last}, \gamma) \models post_1 \wedge post_2$ holds.

Proof. Suppose $\sigma \propto \sigma^1 \parallel \sigma^2$ where $\sigma^1 \in cp[P]$ and $\sigma^2 \in cp[Q]$. From lemma 3 and the hypotheses $\models (rely \vee guar_1) \rightarrow rely_2$ and $\models (rely \vee guar_2) \rightarrow rely_1$, it follows that σ^1 and σ^2 belong to $A(pre, \ rely_1)(\gamma)$ and $A(pre, \ rely_2)(\gamma)$ respectively. By definition 1, $len(\sigma^i) < \infty$ and $Pr(\sigma^i_{last}) = E$ for $i = 1, 2$. Therefore, $(\sigma^1_{last}, \gamma) \models post_1$ and $(\sigma^2_{last}, \gamma) \models post_2$ hold. By definition 1 again, $(\sigma_{last}, \gamma) \models post_1 \wedge post_2$ holds. $\Box$

It follows from Lemmas 4 and 5 that the parallel composition rule is sound.

     Next we discuss the completeness issue. The completeness theorem says that for any program P and specification $(pre, rely, guar, post)$, if $\models P \ \underline{sat} \ (pre, rely, guar, post)$, then $\vdash P \ \underline{sat} \ (pre, rely, guar, post)$.

     The proof of the completeness theorem proceeds by induction on the structure of the program concerned. The basic idea is that if P is a compound statement $P_1 \& P_2$ (here $\&$ is a program combinator, and is called the main combinator of

P_1 and P_2), we first find two suitable sub-specifications $(pre_1, rely_1, guar_1, post_1)$ and $(pre_2, rely_2, guar_2, post_2)$, such that it follows from $\models P$ $\underline{sat}$ $(pre, rely, guar, post)$ that $\models P_1$ $\underline{sat}$ $(pre_1, rely_1, guar_1, post_1)$ and $\models P_2$ $\underline{sat}$ $(pre_2, rely_2, guar_2, post_2)$ hold. By induction, we obtain $\vdash P_1$ $\underline{sat}$ $(pre_1, rely_1, guar_1, post_1)$ and $\vdash P_2$ $\underline{sat}$ $(pre_2, rely_2, guar_2, post_2)$. When these two sub-specifications are suitably chosen, we can derive $\vdash P$ $\underline{sat}$ $(pre, rely, guar, post)$ using the appropriate proof rules.

However, for the case that the main combinator is parallel composition, this simplistic approach is not enough, because the required sub-specifications cannot always be directly expressed. The solution is to introduce an auxiliary variable, transforming $P_1 \parallel P_2$ into $P_1^* \parallel P_2^*$, and use the above simple strategy to show that it satisfies a related specification $(pre^*, rely^*, guar, post)$. Assertions pre^* and $rely^*$ are chosen in such a way that $\vdash P_1 \parallel P_2$ $\underline{sat}$ $(pre, rely, guar, post)$ follows from the auxiliary variable rule.

The use auxiliary variables to increase the expressive powers of the assertions was first discovered by Owicki and Gries, and they also pointed out that in the completeness proof, it is sufficient to use a special auxiliary variable which records the history of the execution. The auxiliary variable is therefore called a history variable. In an early attempt [Stø90], Stølen adapted the history variable construction to prove completeness of his system (which also contains an auxiliary variable rule). Although there are certain minor deficiencies in the proof, Stølen's system is indeed complete.

Below we give a new completeness proof for the rely-guarantee method. The main idea of the proof comes from Stirling's brief remark in [Sti88] that his system is complete relative to Owicki & Gries' system, while the latter one is shown to be relatively complete, in e.g. [Apt81]. Our proof is simpler than the one in [Stø90], but more importantly, it reveals the connection between the Owicki & Gries method and the rely-guarantee method at the completeness level.

Let h denote the history variable, then its value is a sequence of the form $(i_1, \mu_1)(i_2, \mu_2)...(i_n, \mu_n)$, in which i_k is an integer taking the value 1, 2 or 3, denoting whether the transition in question is from P_1, P_2 or the environment respectively, and μ_k is the state at the start of the transition. The pair (i_k, μ_k) is called an i_k record. Process P_i is transformed into P_i^*, by replacing successively

1. each assignment $\bar{x} := \bar{e}$ which is a normal subprogram (not a subprogram of an await statement) by $\bar{x}, h := \bar{e}, h^\wedge(i, y)$, and
2. each await statement **await** b **then** P **end** by **await** b **then** $h := h^\wedge(i, y); P$ **end**, where $h^\wedge(i, y)$ stands for appending (i, y) to h. Execution of boolean tests are not recorded in the history, because they do not change the state. We define

$$pre^*(y, h, y_0) \overset{\text{def}}{=} pre(y, y_0) \wedge h = \langle \rangle$$
$$rely^*((y, h), (y', h'), y_0) \overset{\text{def}}{=} (rely(y, y', y_0) \wedge h' = h^\wedge(3, y)) \vee (y' = y \wedge h' = h)$$

The $rely_i$ and $guar_i$ conditions that we construct use an assertion $PRE_i[R]$, which characterises the following set of states:

$$\{\eta \mid \exists P_1^1, \ldots, P_1^k, P_2^1, \ldots, P_2^k, \eta_0, \ldots, \eta_{k-1}, \gamma, \delta_1, \ldots, \delta_k.$$
$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{\delta_1}{\to} \langle P_1^1 \parallel P_2^1, \eta_1 \rangle \overset{\delta_2}{\to} \cdots \overset{\delta_k}{\to} \langle P_1^k \parallel P_2^k, \eta \rangle \in A(pre^*, rely^*)(\gamma),$$
$$\text{where } R = P_i^k\}$$

The states are from those configurations, whose program parts are R, reached by

computations of $P_1^* \parallel P_2^*$, in which the initial state satisfies pre^* and every environment transition satisfies $rely^*$. In this paper, we do not discuss the particular power a language needs to express $PRE_i[R]$; in other words, we prove what is usually called semantical completeness.

The above notation is rather lengthy, so as an abbreviation, we shall write $\langle P, \eta \rangle \overset{*}{\Rightarrow} \langle P', \eta' \rangle \in A(pre, rely)$, or simply $\langle P, \eta \rangle \overset{*}{\Rightarrow} \langle P', \eta' \rangle$ when the former is understood from the context, for

$$\exists\, P_1, \ldots, P_{k-1}, \eta_1, \ldots, \eta_{k-1}, \gamma, \delta_1, \ldots, \delta_k.$$
$$\langle P, \eta \rangle \overset{\delta_1}{\to} \langle P_1, \eta_1 \rangle \overset{\delta_2}{\to} \cdots \overset{\delta_{k-1}}{\to} \langle P_{k-1}, \eta_{k-1} \rangle \overset{\delta_k}{\to} \langle P', \eta' \rangle \in A(pre,\ rely)(\gamma)$$

Assertion $PRE_i[R]$ can now be simply written as:

$$\exists\, \eta_0, P_1^k, P_2^k. \langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1^k \parallel P_2^k, \eta \rangle \in A(pre^*,\ rely^*)(\gamma)$$
$$\text{where } R = P_i^k$$

For any program P, let $\mathscr{P}(P)$ be the set of all the programs that could arise in the configurations during the execution of P (including the empty program E). In the rest of this section, γ is again an arbitrary logical valuation. For a state η which satisfies both $PRE_1[R_1]$ and $PRE_2[R_2]$, it follows from the definition that there exist P_1' and P_2' such that $\langle R_1 \parallel P_2', \eta \rangle$ and $\langle P_1' \parallel R_2, \eta \rangle$ are reachable. Due to the structure of history variable, we have in fact a stronger result, that is, R_1 and R_2 can be reached at the same time (namely, $\langle R_1 \parallel R_2, \eta \rangle$ is also reachable).

Lemma 6. (Merging lemma) For any states η, η_0 and $P_1' \in \mathscr{P}(P_1^*)$, $P_2' \in \mathscr{P}(P_2^*)$, if $(\eta, \gamma) \models PRE_i[R]$ and $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel P_2', \eta \rangle \in A(pre^*,\ rely^*)(\gamma)$, then $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel R, \eta \rangle \in A(pre^*,\ rely^*)(\gamma)$, when $i = 2$, and, similarly when $i = 1$, $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle R \parallel P_2', \eta \rangle \in A(pre^*,\ rely^*)(\gamma)$.

Proof. We give a proof for $i = 2$. From $(\eta, \gamma) \models PRE_2[R]$, it follows that $\exists \eta_0, P_1^1$. $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1^1 \parallel R, \eta \rangle \in A(pre^*,\ rely^*)(\gamma)$. We prove the lemma by induction on the number of i records in η.

Base step: assume there are no i records in η, then the transitions from P_i^* do not change the program variables (they can only be boolean tests in this case), and the construction is straightforward.

Induction step: assume the lemma holds for the case $n = k$, now consider the case $n = k + 1$. Suppose the history variable h in state η has the value of the form $\theta'^\frown(i, \bar{\mu})^\frown\theta''$, where θ'' does not contain any i records, then there exist $\eta', \eta'', P_1', P_2'', P_2''', P_1^2, P_2^2, P_2^3$ with $\eta'(h) = \theta'$ and $\eta''(h) = \theta'^\frown(i, \bar{\mu})$, such that

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1'' \parallel P_2'', \eta' \rangle \overset{c}{\to} \langle P_1'' \parallel P_2''', \eta'' \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel P_2', \eta \rangle$$

and

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1^2 \parallel P_2^2, \eta' \rangle \overset{c}{\to} \langle P_1^2 \parallel P_2^3, \eta'' \rangle \overset{*}{\Rightarrow} \langle P_1^1 \parallel R, \eta \rangle$$

By the induction hypothesis,

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1'' \parallel P_2^2, \eta' \rangle$$

Thus $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1'' \parallel P_2^2, \eta' \rangle \overset{c}{\to} \langle P_1'' \parallel P_2^3, \eta'' \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel R, \eta \rangle$ $\square$

Theorem 2. (completeness) The proof system in this section is semantically complete.

Proof. By induction on the program structure. Here, we only discuss the case that the main program combinator is the parallel composition.

Suppose $\models P_1 \parallel P_2$ _sat_ (pre, $rely$, $guar$, $post$), then we have $\models P_1^* \parallel P_2^*$ _sat_ (pre^*, $rely^*$, $guar$, $post$). Let

$$rely_i \stackrel{\text{def}}{=} \bigwedge_{R \in \mathscr{P}(P_i^*)} (PRE_i[R] \to PRE_i[R]')$$
$$guar_i \stackrel{\text{def}}{=} rely_j \wedge guar, \quad j \neq i$$
$$post_i \stackrel{\text{def}}{=} PRE_i[E]$$

Before proving $\models P_i^*$ _sat_ (pre^*, $rely_i$, $guar_i$, $post_i$), we relate these assertions to the ones used in the completeness proof for the Owicki & Gries method. We have stated that the rely–guarantee method can be regarded as a compositional reformulation of the Owicki & Gries method, and now we can see how this is reflected in the completeness proof. In the completeness proof for the Owicki & Gries method, the pre-condition for each subprogram R in the local verification is essentially the same as $PRE_i[R]$, and the interference freedom test requires proving for each atomic statement (assignment or await statement) Z in P_j^* that

$$\{PRE_i[R] \wedge PRE_j[Z]\} \, Z \, \{PRE_i[R]\}$$

This interference freedom test can be interpreted as saying that if R is reachable in the current state, then it is still reachable in the state after a transition from its environment. For process P_i^*, if the environment transitions satisfy $rely_i$, then by its definition, for any $R \in \mathscr{P}(P_i^*)$, $PRE_i[R]$ is satisfied after the transitions when it is also satisfied before. Since the guarantee–condition of one process implies the rely–condition of the other process, the same interference freedom is ensured by the compositional method. $\square$

Lemma 7. For any $R \in \mathscr{P}(P_i^*)$, $\models (rely^* \wedge PRE_i[R]) \to PRE_i[R]'$

Proof. As before, we assume in the proof that $i = 2$.
For any η, η' such that $(\eta, \eta', \gamma) \models rely^*$ and $(\eta, \gamma) \models PRE_i[R]$ hold, we have to show that $(\eta', \gamma) \models PRE_i[R]$ holds. By the definition of $PRE_i[R]$, there exist η_0 and P_1' such that $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \stackrel{*}{\Rightarrow} \langle P_1' \parallel R, \eta \rangle$. It follows from the transition rule $\langle P_1' \parallel R, \eta \rangle \stackrel{e}{\to} \langle P_1' \parallel R, \eta' \rangle$ that $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \stackrel{*}{\Rightarrow} \langle P_1' \parallel R, \eta' \rangle$ holds, thus $(\eta', \gamma) \models PRE_i[R]$ holds by the definition. $\square$

Lemma 8. $\models (rely^* \vee guar_j) \to rely_i$ for $i \neq j$, and $\models guar_1 \vee guar_2 \to guar$.

Proof. From lemma 7, one immediately gets

$$\bigwedge_{R \in \mathscr{P}(P_i^*)} (rely^* \wedge PRE_i[R]) \to PRE_i[R]'$$
$$= \bigwedge_{R \in \mathscr{P}(P_i^*)} rely^* \to (PRE_i[R] \to PRE_i[R]')$$
$$= rely^* \to \bigwedge_{R \in \mathscr{P}(P_i^*)} (PRE_i[R] \to PRE_i[R]')$$
$$= rely^* \to rely_i$$

By the definition of $guar_i$, $\forall j \neq i.guar_j \to rely_i$. Therefore, $\models (rely^* \vee guar_j) \to rely_i$ holds. It follows directly from the definition that $\models guar_1 \vee guar_2 \to guar$ holds. $\square$

Lemma 9. For any n and $\langle P_i^n, \eta_n \rangle$, if there exist $P_i^0, P_i^1, \ldots, P_i^{n-1}$, $\eta_0, \ldots, \eta_{n-1}$, $\delta_1, \ldots, \delta_n$ such that $P_i^0 = P_i^*$ and $\langle P_i^0, \eta_0 \rangle \overset{\delta_1}{\to} \langle P_i^1, \eta_1 \rangle \overset{\delta_2}{\to} \cdots \overset{\delta_n}{\to} \langle P_i^n, \eta_n \rangle \in A(pre^*, rely_i)(\gamma)$, then $(\eta_n, \gamma) \models PRE_i[P_i^n]$ holds.

Proof. By induction on n. Without loss of generality, we assume in the proof that $i = 2$.

Base step: n=0, then $P_2^n = P_2^*$ and obviously $(\eta_0, \gamma) \models PRE_i[P_2^n]$.

Induction step: assume the lemma holds for $n = k$, consider the case $n = k + 1$.

If $\langle P_2^*, \eta_0 \rangle \overset{\delta_1}{\to} \langle P_2^1, \eta_1 \rangle \overset{\delta_2}{\to} \ldots \overset{\delta_k}{\to} \langle P_2^k, \eta_k \rangle \overset{\delta_{k+1}}{\to} \langle P_2^{k+1}, \eta_{k+1} \rangle$, then by the induction hypothesis $(\eta_k, \gamma) \models PRE_i[P_2^k]$. If $\delta_{k+1} = e$, then $P_2^{k+1} = P_2^k$ and by the definition of $rely_2$, $(\eta_{k+1}, \gamma) \models PRE_i[P_2^{k+1}]$ follows. If $\delta_{k+1} = c$, then from $(\eta_k, \gamma) \models PRE_i[P_2^k]$, there exists P_1' such that $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel P_2^k, \eta_k \rangle$. Because $\langle P_2^k, \eta_k \rangle \overset{c}{\to} \langle P_2^{k+1}, \eta_{k+1} \rangle$, we have $\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel P_2^{k+1}, \eta_{k+1} \rangle$. Thus, $(\eta_{k+1}, \gamma) \models PRE_i[P_2^{k+1}]$ holds. $\square$

Lemma 10. For any $\langle P_i^*, \eta_0 \rangle \overset{\delta_1}{\to} \langle P_i^1, \eta_1 \rangle \overset{\delta_2}{\to} \cdots \overset{\delta_k}{\to} \langle P_i^k, \eta_k \rangle \overset{c}{\to} \langle P_i^{k+1}, \eta_{k+1} \rangle \in A(pre^*, rely_i)(\gamma)$, $(\eta_k, \eta_{k+1}, \gamma) \models guar_i$ holds.

Proof. By the definition of $guar_i$, we are required to show that $(\eta_k, \eta_{k+1}, \gamma) \models guar$ and $(\eta_k, \eta_{k+1}, \gamma) \models rely_j$ ($j \neq i$) hold. Assume $i = 2$ in the proof.

By lemma 9, $(\eta_k, \gamma) \models PRE_2[P_2^k]$, thus there exists P_1' such that

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel P_2^k, \eta_k \rangle$$

therefore

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle P_1' \parallel P_2^k, \eta_k \rangle \overset{c}{\to} \langle P_1' \parallel P_2^{k+1}, \eta_{k+1} \rangle$$

From $\models P_1^* \parallel P_2^*$ <u>sat</u> $(pre^*, rely^*, guar, post)$, it follows that $(\eta_k, \eta_{k+1}, \gamma) \models guar$ holds.

To show $(\eta_k, \eta_{k+1}, \gamma) \models rely_1$, assume that $(\eta_k, \gamma) \models PRE_1[R]$ holds for $R \in \mathscr{P}(P_1^*)$, we have to prove $(\eta_{k+1}, \gamma) \models PRE_1[R]$. By the merging lemma,

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle R \parallel P_2^k, \eta_k \rangle$$

and therefore it follows from the transition

$$\langle P_2^k, \eta_k \rangle \overset{c}{\to} \langle P_2^{k+1}, \eta_{k+1} \rangle$$

that

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle R \parallel P_2^{k+1}, \eta_{k+1} \rangle$$

holds. Hence, $(\eta_{k+1}, \gamma) \models PRE_1[R]$ follows from the definition of $PRE_1[R]$. $\square$

Lemma 11. If $\langle P_i^*, \eta_0 \rangle \overset{\delta_1}{\to} \langle P_i^1, \eta_1 \rangle \overset{\delta_2}{\to} \cdots \overset{\delta_n}{\to} \langle E, \eta_n \rangle \in A(pre^*, rely_i)(\gamma)$, then $(\eta_n, \gamma) \models post_i$.

Proof. $\langle P_i^*, \eta_0 \rangle \overset{\delta_1}{\to} \langle P_i^1, \eta_1 \rangle \overset{\delta_2}{\to} \cdots \overset{\delta_n}{\to} \langle E, \eta_n \rangle \in A(pre^*, rely_i)(\gamma)$ implies by lemma 9, that $(\eta_n, \gamma) \models PRE_i[E]$, that is, $(\eta_n, \gamma) \models post_i$. $\square$

Summarising, we have so far shown

$$\models (rely^* \lor guar_j) \rightarrow rely_i \qquad\qquad j \neq i,$$
$$\models guar_1 \lor guar_2 \rightarrow guar$$
$$\models P_i^* \underline{sat} \ (pre^*, rely_i, \ guar_i, \ post_i)$$

Hence, by the relative completeness assumption, the induction hypothesis and the parallel rule, we have

$$\vdash P_1^* \parallel P_2^* \underline{sat} \ (pre^*, rely^*, \ guar, \ post_1 \land post_2)$$

Lemma 12. $\models post_1 \land post_2 \rightarrow post$

Proof. Suppose $(\eta, \gamma) \models post_1 \land post_2$ holds. From $(\eta, \gamma) \models post_1$, we know that there exist η_0 and P_2' such that

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle E \parallel P_2', \eta \rangle$$

Because $(\eta, \gamma) \models post_2$, namely $(\eta, \gamma) \models PRE_2[E]$ holds, by the merging lemma,

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle E \parallel E, \eta \rangle$$

From the assumption $\models P_1^* \parallel P_2^* \underline{sat} \ (pre^*, rely^*, \ guar, \ post)$, it follows that $(\eta, \gamma) \models post$. $\square$

By the relative completeness assumption and the consequence rule, we now have

$$\vdash P_1^* \parallel P_2^* \underline{sat} \ (pre^*, rely^*, \ guar, \ post)$$

and finally by the auxiliary variable rule, we get

$$\vdash P_1 \parallel P_2 \underline{sat} \ (pre, rely, \ guar, \ post)$$

4. Verification of Deadlock Freedom

In the preceding sections, we have ignored the problem of deadlock. The statement **await** b **then** P **end** provides a means of synchronisation, and it is only successfully executed when the boolean condition b is true. When condition b does not hold, the process is blocked, and can only become active again when another process sets this condition to true. This kind of waiting is essential for the correct functioning of some algorithms. However, if the synchronisation statements are not used with care, it might happen that at a certain point all the processes of a system are blocked. Then no process can make a move anymore, and in this case the system is said to be *deadlocked*. Partial correctness does not exclude deadlocked computations, because only successfully terminated states are required to satisfy post-conditions. Deadlock is clearly undesirable, for all the concurrent programs that we are aware of are either supposed to terminate normally or run forever, that is, they should never end up in deadlock. This section discusses verification of deadlock freedom, that is, deadlock does not occur.

4.1. Specification and Correctness

Let us first review the method of Owicki & Gries. To simplify the exposition, we assume that the program has only two processes. To use the Owicki & Gries

method, one first identifies the states in which it is possible for the two sub-
processes to be blocked; assume $wait_i$ is the predicate characterising the possible
blocking states of the corresponding process. When the program is deadlocked in
a state η, either both processes are blocked, or one is blocked and the other one
has terminated. Hence the following holds:

$$(\eta, \gamma) \models (wait_1 \wedge post_2) \vee (wait_2 \wedge post_1) \vee (wait_1 \wedge wait_2)$$

where γ is the logical state. Therefore, if one can prove

$$(wait_1 \wedge post_2) \vee (wait_2 \wedge post_2) \vee (wait_1 \wedge wait_2) = false \qquad \text{DF1}$$

then the system is deadlock free.

Stølen [Stø90] observed one can verify deadlock freedom compositionally by
including blocking information in the specifications. He suggested augmenting
Jones' specification quadruple by another component $wait$; a program is correct
with respect to such a specification if it is only blocked in states satisfying $wait$.
A predicate actually equivalent to DF1 appeared in Stølen's parallel composition
rule as the additional proof obligation. As a matter of fact, both Jones' suggestion
of specifying interference and Stølen's suggestion of specifying blocking represent
a typical strategy in formulating compositional rules from non–compositional
ones: relevant information obtained from the environment process with the help
of its structure in the non–compositional method is made independent from the
structure and included as part of a specification.

We followed Stølen's suggestion of augmenting Jones's specification, but
looked at the problem from another angle [XuH91, Xu92]. Consider Owicki
& Gries' deadlock freedom condition DF1 again. It is easy to show that DF1 is
equivalent to

$$(post_1 \rightarrow \neg wait_2) \wedge (post_2 \rightarrow \neg wait_1) \wedge (\neg wait_1 \vee \neg wait_2)$$

The predicate $\neg wait_i$ describes the set of states in which the process is not blocked,
that is, either the process has terminated or is active. Let us rename $\neg wait_i$ by
run_i, then the deadlock freedom condition becomes

$$(post_1 \rightarrow run_2) \wedge (post_2 \rightarrow run_2) \wedge (run_1 \vee run_2) \qquad \text{DF2}$$

It is easy to understand condition DF2: deadlock cannot occur if, 1) when one
process has terminated, the other one is not blocked, and, 2) at any time, at
least one process is not blocked. Note point 2) alone is not sufficient to ensure
deadlock freedom, because it does not rule out the possibility that one process has
terminated while the other one is blocked — point 1) is needed exactly for this
purpose. Notice that in DF2, the post-condition of one process implies the run-
condition of another process, that is, they also form an assumption/commitment
pair. This is the reason why we favour including run (rather than $wait$) in a
specification.

A specification is now of the form

$$(pre, \; rely, \; run, \; guar, \; post)$$

A program P satisfies such a specification if, under the same assumptions as in
Section 3.1 concerning pre and $rely$, in addition

$$P \text{ is not blocked if } run \text{ holds}$$

Formally, the program is not blocked after a computation σ if and only if

$$len(\sigma) = \infty \vee Pr(\sigma_{last}) = E \vee \sigma_{last} \overset{c}{\to}$$

Let σ and γ be a computation and an arbitrary logical valuation respectively, and define

$$C(run)(\gamma) \overset{\text{def}}{=} \{\sigma \mid \text{if } len(\sigma) < \infty \text{ and } (\sigma_{last}, \gamma) \models run,$$
$$\text{then } Pr(\sigma_{last}) = E \text{ or } \sigma_{last} \overset{c}{\to}\}$$

$$C(run, guar, post)(\gamma) \overset{\text{def}}{=} C(run)(\gamma) \cap C(guar, post)(\gamma)$$

The satisfaction relation is defined as:

$\models P \underline{\ sat\ } (pre,\ rely,\ run,\ guar,\ post)$
$\overset{\text{def}}{=} \forall \gamma.\, A(pre, rely)(\gamma) \cap cp[P] \subseteq C(run, guar, post)(\gamma)$

An example may help understand this definition. Take P as the program

await $x = 1$ **then** $x := 2$ **end**

Clearly, when $x = 1$ holds, P is not blocked. The following correctness formula is valid:

$P \underline{\ sat\ } (true,\ true,\ x = 1,\ true,\ true)$

This does not mean that the process cannot be blocked at all, and as a matter of fact, the following computation of P:

$$\sigma : \quad \langle P, 0 \rangle \overset{e}{\to} \langle P, 2 \rangle \overset{e}{\to} \langle P, 1 \rangle \overset{e}{\to} \langle P, 5 \rangle$$

indeed ends in a blocked state, but it does not invalidate the correctness formula because the last state of σ does not satisfy the run-condition $x = 1$, and therefore the computation still belongs to $C(run)(\gamma)$. The following computation of P does not invalidate the correctness formula either:

$$\tau : \quad \langle P, 0 \rangle \overset{e}{\to} \langle P, 2 \rangle \overset{e}{\to} \langle P, 1 \rangle \overset{e}{\to} \langle P, 5 \rangle \overset{e}{\to} \langle P, 1 \rangle$$

since, being a non-blocked computation (because $\langle P, 1 \rangle \overset{c}{\to}$ holds), it clearly belongs to $C(run)(\gamma)$. On the other hand, P does not satisfy the specification:

$(true,\ true,\ x = 5,\ true,\ true)$

as, for instance, the first of the above two computations of P does not belong to $C(run)(\gamma)$.

4.2. Proof Rules

The proof system given in section 3.2 has to be modified to verify additionally deadlock freedom, but only the changes to the await and parallel composition rules are nontrivial.

Await axiom

$$\frac{\begin{array}{l} pre \underline{\ stable\ when\ } rely \\ post \underline{\ stable\ when\ } rely \\ P \underline{\ sat\ } (pre \wedge b \wedge v_0 = y,\ y' = y,\ true,\ true,\ \ guar[v_0/y,\ y/y'] \wedge post) \end{array}}{\textbf{await } b \textbf{ then } P \textbf{ end } \underline{\ sat\ } (pre,\ rely,\ pre \to b,\ guar,\ post)}$$

where v_0 is a vector of fresh logical variables.

The await statement is not blocked if b holds. Because *pre* holds initially and it is stable with respect to *rely*, it holds before the statement is executed. This ensures that under the run-condition $pre \rightarrow b$, the await statement is not blocked. The await statement

await $x > 0$ **then** $x := x - 1$ **end**

satisfies

$$(x \geq 0, \ x \geq 0 \rightarrow x' \geq 0, \ x \geq 0 \rightarrow x > 0, \ x' \leq x, \ x \geq 0)$$

Parallel composition rule

$$\begin{array}{l}
(post_1 \rightarrow run_2) \wedge (post_2 \rightarrow run_1) \wedge (run_1 \vee run_2) \\
(rely \vee guar_1) \rightarrow rely_2 \\
(rely \vee guar_2) \rightarrow rely_1 \\
(guar_1 \vee guar_2) \rightarrow guar \\
P \ \underline{sat} \ (pre, \ rely_1, \ run \wedge run_1, \ guar_1, \ post_1) \\
\underline{Q \ \underline{sat} \ (pre, \ rely_2, \ run \wedge run_2, \ guar_2, \ post_2)} \\
\hline
P \parallel Q \ \underline{sat} \ (pre, \ rely, \ run, \ guar, \ post_1 \wedge post_2)
\end{array}$$

We now argue that it is not possible for $P \parallel Q$ to be blocked, under the premises given in the rule and the assumptions in the specification. Suppose this is not the case, namely, that $P \parallel Q$ is blocked in a state satisfying *run*. We demonstrate that it leads to a contradiction as follows. There are two possibilities for $P \parallel Q$ to be blocked: either one of the processes has terminated and the other one is blocked, or both of them are blocked. In the first case, suppose P has terminated and Q is blocked. Hence, $post_1$ holds by the premise $P \ \underline{sat} \ (pre, \ rely_1, \ run \wedge run_1, \ guar_1, \ post_1)$, and from the premise that $post_1 \rightarrow run_2$ is valid, it follows that run_2 holds too, thus, $run \wedge run_2$ holds. However, from the specification of Q, $run \wedge run_2$ ensures that Q is not blocked, therefore leading to a contradiction. In the second case, it follows from the premise that $run_1 \vee run_2$ is valid and the assumption that *run* holds in the blocked state, at least one of P and Q is not blocked due to their specifications. Again, we have a contradiction.

Consider the program

await $x > 0$ **then** $x := 2$ **end** $\parallel x := 1$

One expects that it satisfies

$$(x = 0, \ x' = x, \ true, \ true, \ x = 2)$$

This can be deduced as follows. First, it is easy to prove that program

await $x > 0$ **then** $x := 2$ **end**

(by the await and consequence rules) satisfies

$$\begin{array}{ll}
pre : & x = 0 \\
rely : & x' = x \vee (x = 0 \wedge x' = 1) \\
run : & x > 0 \\
guar : & x' = x \vee (x > 0 \wedge x' = 2) \\
post : & x = 2
\end{array}$$

and program $x := 1$ satisfies

$$pre : \quad x = 0$$
$$rely : \quad x' = x \vee (x > 0 \wedge x' = 2)$$
$$run : \quad true$$
$$guar : \quad x' = x \vee (x = 0 \wedge x' = 1)$$
$$post : \quad x = 1 \vee x = 2$$

Second, applying the parallel compositional rule, we can establish the original correctness formula.

4.3. Soundness and Completeness

Theorem 3. (soundness and completeness) The extended proof system is sound and semantically complete.

We only mention the non-trivial changes in the proofs.

Let γ be an arbitrary logical valuation in the following proofs concerning soundness.

Soundness of the await rule. We only show that any finite computation σ from the await statement is not blocked if the run-condition $pre \rightarrow b$ holds in the last state. Suppose σ is of the form $\sigma_0 \xrightarrow{e} \sigma_1 \xrightarrow{e} \cdots \xrightarrow{e} \sigma_n$, then it follows from pre <u>stable when</u> $rely$ that $(\sigma_n, \gamma) \models pre$. This, together with $(\sigma_n, \gamma) \models pre \rightarrow b$, implies that $(\sigma_n, \gamma) \models b$, and therefore the await statement is not blocked.

Soundness of the parallel rule. Now in addition, we have as part of the assumption

$$\models (post_1 \rightarrow run_2) \wedge (post_2 \rightarrow run_1) \wedge (run_1 \vee run_2)$$

and P and Q are not blocked if $run \wedge run_1$ and $run \wedge run_2$ hold respectively.

Lemma 13. Under these assumptions, $A(pre, rely)(\gamma) \cap cp[P \parallel Q] \subseteq C(run)(\gamma)$.

Proof. Suppose σ is in $A(pre, rely)(\gamma) \cap cp[P \parallel Q]$, but not in $C(run)(\gamma)$, then σ is finite, $\sigma_{last} \neq E$ and $\neg \sigma_{last} \xrightarrow{c}$. Let $len(\sigma) = n$. By the assumption that $\sigma \notin C(run)(\gamma)$, $(\sigma_n, \gamma) \models run$. There are two possibilities, either one of P and Q has terminated, but the other is blocked, or both are blocked. We only prove the first case, and the second one is simpler. Suppose Q is the one which has terminated. From lemma 1, there exist σ^1 and σ^2 belonging to $cp[P]$ and $cp[Q]$ respectively, such that $\sigma \propto \sigma^1 \parallel \sigma^2$. By lemma 3 and the premise $\models Q$ <u>sat</u> $(pre, rely_2, run \wedge run_2, guar_2, post_2)$, we have $(\sigma_n, \gamma) \models post_2$. It then follows from the premise $\models post_2 \rightarrow run_1$ that $(\sigma_n, \gamma) \models run_1$, thus $(\sigma_n, \gamma) \models run_1 \wedge run$, and this contradicts the premise P <u>sat</u> $(pre, rely_1, run \wedge run_1, guar_1, post_1)$, which implies that P is not blocked when $run \wedge run_1$ is satisfied. $\square$

Completeness of the parallel composition rule. We need to find run_i which is to ensure that the corresponding program P_i^* can proceed when it is in a state satisfying $run \wedge run_i$. First let us examine when a non–terminated program P can proceed. This can be easily described inductively by $r(P)$:

$$r(P) \overset{\text{def}}{=} true, \qquad \text{if } P = \bar{x} := \bar{e}, \text{ while } b \text{ do } S \text{ od,}$$
$$\qquad\qquad\qquad\qquad \text{if } b_1 \to P_1 \square \cdots \square b_n \to P_n \text{ fi}$$
$$\overset{\text{def}}{=} b, \qquad\quad \text{if } P = \textbf{await } b \textbf{ then } S \textbf{ end}$$
$$\overset{\text{def}}{=} r(R), \qquad\; \text{if } P = R;S$$
$$\overset{\text{def}}{=} r(R) \vee r(S), \quad \text{if } P = R \parallel S$$

To this end, we can define

$$run_i \overset{\text{def}}{=} \bigwedge\nolimits_{R \in \widehat{\mathscr{P}}(P_i^*)} ((PRE_i[R] \wedge run) \to r(R))$$

where $\widehat{\mathscr{P}}(P_i^*) \overset{\text{def}}{=} \mathscr{P}(P_i^*)\backslash\{E\}$ (that is, the set of all proper programs that could arise during the execution of P). It is easy to see that P_i^* will not be blocked when $run \wedge run_i$ holds. In addition, the following lemma has to be established as part of the proof obligations.

Lemma 14. $\models (post_1 \to run_2) \wedge (post_2 \to run_1) \wedge (run_1 \vee run_2)$

Proof. First we show that $\models post_1 \to run_2$. Suppose this is not the case, then there exists a state η and a logical valuation γ such that $(\eta, \gamma) \models post_1$ and $(\eta, \gamma) \not\models run_2$. By the definition of run_2, there exists $R \in \widehat{\mathscr{P}}(P_2^*)$ such that $(\eta, \gamma) \models run$, $(\eta, \gamma) \models PRE_2[R]$, and $(\eta, \gamma) \not\models r(R)$. It follows from the merging lemma that

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle E \parallel R, \eta \rangle$$

but $\neg\langle R, \eta \rangle \overset{c}{\to}$. This contradicts the assumption $\models P_1^* \parallel P_2^*$ $\underline{sat}$ $(pre^*, rely^*, run, guar, post)$.

The case for $\models post_2 \to run_1$ is the same, so what remains is to prove $\models run_1 \vee run_2$. Assume this is not the case, then there exist a state η and a logical valuation γ such that $(\eta, \gamma) \not\models run_1$ and $(\eta, \gamma) \not\models run_2$, therefore, there exist $R \in \widehat{\mathscr{P}}(P_1^*)$ and $Z \in \widehat{\mathscr{P}}(P_2^*)$ such that $(\eta, \gamma) \models run$, $(\eta, \gamma) \models PRE_1[R]$ and $(\eta, \gamma) \models PRE_2[Z]$, but $(\eta, \gamma) \not\models r(R)$ and $(\eta, \gamma) \not\models r(Z)$. Consequently, $(\eta, \gamma) \not\models r(R) \vee r(Z)$. By the merging lemma,

$$\langle P_1^* \parallel P_2^*, \eta_0 \rangle \overset{*}{\Rightarrow} \langle R \parallel Z, \eta \rangle$$

but $\neg\langle R \parallel Z, \eta \rangle \overset{c}{\to}$. This again contradicts the assumption $\models P_1^* \parallel P_2^*$ $\underline{sat}$ $(pre^*, rely^*, run, guar, post)$. $\square$

5. Total and Weakly Total Correctness

In the previous sections, the commitment part of a specification allows infinite computations provided the guarantee-condition is satisfied. Sometimes, it is desirable to require a program to be *convergent*, that is, the program cannot loop forever if the assumption of the specification is satisfied. A program is *weakly total correct*, if it is partially correct and convergent. A program is *totally correct*, if it is weakly total correct and deadlock free.

5.1. Specifications, Correctness and Proof Rules

To establish weakly total correctness, the specification format for partial correctness

$(pre, rely, guar, post)$

is used; to verify total correctness, the specification format for partial correctness and deadlock freedom

$(pre, rely, run, guar, post)$

is adopted. In both cases, the interpretation of a correctness formula has the extra commitment that there are not infinitely many component steps. Let σ be a computation, define

$$C_l \stackrel{\text{def}}{=} \{\sigma \mid \sigma \text{ has only a finite number of } c\text{–transitions.}\}$$

Considering total correctness, for example, we define

$$C'(run, guar, post)(\gamma) \stackrel{\text{def}}{=} C(run, guar, post)(\gamma) \cap C_l,$$

and modify the satisfaction relation as follows:

$$\models P \ \underline{sat} \ (pre, rely, run, guar, post)$$
$$\stackrel{\text{def}}{=} \forall \gamma. A(pre, rely)(\gamma) \cap cp[P] \subseteq C'(run, guar, post)(\gamma)$$

Since the only program structure which can cause divergence is iteration, it suffices to modify the corresponding rule.

Iteration rule

$(Inv(\alpha) \wedge \alpha > 0) \to b$
$Inv(0) \to \neg b$
$(Inv(\alpha) \wedge rely) \to \exists \beta \leqslant \alpha. Inv(\beta)'$
$\underline{P \ \underline{sat} \ (Inv(\alpha) \wedge \alpha > 0, \ rely, \ run, \ guar, \ \exists \beta < \alpha. Inv(\beta))}$
while b **do** P **od** $\underline{sat}$ $(\exists \alpha. Inv(\alpha), \ rely, \ run, \ guar, \ Inv(0))$

where Inv is the parametrised invariant, and α and β are logical variables ranging over a set of ordinal numbers. As usual, 0 denotes the least ordinal and $<$ denotes the strict ordering of ordinals.

One could also extend the proof system for partial correctness in Section 3 along the same lines. This leads to a system for weakly total correctness.

It was observed first by Back [Bac81] that natural number-valued loop counters are insufficient when unbounded nondeterminism is present. In the rely-guarantee method, although any particular program has only a bounded number of choices at any moment, it is not reasonable to expect the nondeterminism caused by the environment to be bounded too. Consider the following program:

```
while x = 0 ∨ 0 < y do
   if x = 0 then x := 1
            else y := y − 1 fi
od
```

Semantically, it satisfies the following specification:

$pre :$ $x = 0$
$rely :$ $(x' = x) \wedge (x \neq 0 \to y' = y)$
$guar :$ $true$
$post :$ $true$

It is easy to see that when x is set to 1, the value in y is the number of iterations. However, when $x = 0$, the environment can assign to y an arbitrarily large natural

number. Therefore, no natural number can be used as the loop counter before the loop is entered.

Define $Inv(\alpha)$ by

$$Inv(\alpha) \stackrel{\text{def}}{=} (x = 0 \rightarrow \alpha = \omega) \wedge (x \neq 0 \rightarrow \alpha = y)$$

where ω is a constant ordinal bigger than all the natural numbers. The proof obligations consist of showing that the following formulas are valid

$$(Inv(\alpha) \wedge \alpha > 0) \rightarrow (x = 0 \vee 0 < y)$$
$$Inv(0) \rightarrow \neg(x = 0 \vee 0 < y)$$
$$(Inv(\alpha) \wedge rely) \rightarrow \exists \beta \leqslant \alpha. Inv(\beta)'$$

and that **if** $x = 0$ **then** $x := 1$ **else** $y := y - 1$ **fi** satisfies

$$
\begin{aligned}
pre &: \quad Inv(\alpha) \wedge \alpha > 0 \\
rely &: \quad (x' = x) \wedge (x \neq 0 \rightarrow y' = y) \\
guar &: \quad true \\
post &: \quad \exists \beta < \alpha. Inv(\beta)
\end{aligned}
$$

The first two follow directly from the definition of $Inv(\alpha)$; for the third, we can choose $\beta = \alpha$ and the rest is easy to prove by the definition of $Inv(\alpha)$. The correctness formula for the conditional statement is also straightforward.

5.2. Soundness and Completeness

The new iteration rule is shown to be sound and semantically complete in [Xu92], but the details are rather tedious and do not provide any insights into concurrency, and hence are omitted here.

6. Related Work

It is now well realised that parallel computer programs are among the most complicated systems that human beings have ever constructed in terms of the number of interacting components and the degree of interaction. Constructing such systems has proven to be much harder than expected. This paper gives a systematic account of the rely–guarantee method developed along the lines suggested by Cliff Jones [Jon81]. The emphasis has been on compositionality. Using the rely–guarantee method, a number of examples have been worked out in [WoD88, Stø90, Xu92]. Recently, Jones [Jon96] explores how the rely–guarantee idea can be used in an objected–oriented setting.

A number of researchers have incorporated the rely–guarantee ideas into existing full temporal based logics; examples are the works by Abadi and Lamport [AbL95] on TLA, by Collette [Col93] on UNITY, by Moszkowski on ITL [Mos94], and by Jonsson and Tsay [JoT95] on linear-time temporal logic.

A related topic of research concerns the abstraction of the semantic model. In this paper, we concentrate on the proof methods, and have used a simple semantics. To have a more abstract model, the program text should not be part of the observables. A semantics which only records state transition sequences is given in [XuH91], and to further promote abstraction, two closure conditions are suggested. Brookes [Bro93] has studied a basically identical model independently, and has actually shown it is fully abstract with respect to partial correctness.

In [MiC81], Misra and Chandy investigated a rely–guarantee method for another paradigm of concurrency, namely, the one based on message passing. The rely–guarantee methods for shared variable and message passing concurrency were studied independently. But in fact the two paradigms are closely related, and the two parallel composition rules turn out to be special cases of a general rule [XCC94].

Acknowledgements

References

[Apt81]   Apt, K. R.: Recursive assertions and parallel programs. *Acta Informatica*, Springer-Verlag, 1981.

[AbL95]   Abadi, M. and Lamport, L.: Conjoining specifications. *ACM Trans. on Program. Lang. Syst.*, 17(3): 507–534, 1995.

[Bac81]   Back, R. J. R.: Proving total correctness of nondeterministic programs in infinitary logic. *Acta Informatica*, Springer-Verlag, 1981.

[Bro93]   Brookes, S. D.: Full abstraction for a shared variable parallel language. In *Proc. 8th IEEE Int. Symp. on Logic in Computer Science*, 1993.

[Col93]   Collette, P.: Application of the composition principle to Unity–like specifications. In M.-C. Gaudel and J.-P. Jouannaud eds., *Proc. of TAPSOFT 93*, LNCS 668, Springer-Verlag, 1993.

[Coo78]   Cook, S. A.: Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing* 7: 70–90, 1978.

[deRo85]   de Roever, W. P.: The quest for compositionality. in *Proc:IFIP Working Conf. The Role of Abstract Models in Computer Science*. North-Holland, 1985.

[Jon81]   Jones, C. B.: *Development methods for computer programs including a notion of interference*. DPhil. Thesis, Oxford University Computing Laboratory, 1981.

[Jon96]   Jones, C. B.: Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2): 105–121, 1996.

[JoT95]   Jonsson, B. and Tsay, Y.-K.: Reasoning about assumption/guarantee specifications in linear-time temporal logic. In *Proc. of TAPSOFT 95*, LNCS, Springer-Verlag, 1995.

[Lam95]   Lamport, L.: The temporal logic of actions. *ACM Trans. on Program. Lang. Syst.*, 16(3): 872–923, 1995.

[MiC81]   Misra, J. and Chandy, M.: Proofs of networks of processes. *IEEE SE*, 7(4): 417–426, 1981.

[Mos94]   Moszkowski, B.: Some very compositional temporal properties, In *Programming Concepts, Methods and Calculi (A-56)*, E.-R. Olderog (Editor), Elsevier Science B.V. (North-Holland), pp. 307 -326, 1994.

[OwG76]   Owicki, S. and Gries, D.: An axiomatic proof technique for parallel programs. *Acta Inform. 6*, pp. 319 -340, Springer-Verlag, 1976.

[Plo81]   Plotkin, G. D.: A structural approach to operational semantics. Computer Science Department, Aarhus University, Technical Report, DAIMI FN-19,1981.

[Sti88]   Stirling, C.: A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science* 58: 347–359, 1988.

[Stø90]   Stølen, K.: *Development of Parallel Programs on Shared Data-structures*. Ph.D Thesis, Computer Science Department, Manchester University, 1990.

[Stø91a]   Stølen, K.: An attempt to reason about shared-state concurrency in the style of VDM. in S. Prehn and W. J. Toetenel, editors, *Proceedings of VDM 91*, LNCS 551, Springer-Verlag, 1991.

[Stø91b]   Stølen, K.: A method for the development of totally correct shared-state parallel
            programs. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings of CONCUR 91*,
            LNCS 527, Springer-Verlag, 1991.
[Stø92a]   Stølen, K.: Proving total correctness with respect to a fair (share-state) parallel language.
            In *Proceedings of BCS FACS 5th Refinement Workshop* January 1992, London, Springer-
            Verlag.
[Stø92b]   Stølen, K.: Shared-state design modulo weak and strong process fairness. In *Proceedings
            of 5th International Conference on Formal Description Techniques*, October 1992, Perros-
            Guirec, France.
[WoD88]   Woodcock, J. C. P. and Dickinson, B.: Using VDM with Rely and Guarantee-conditions,
            experiences from a real project. In *2nd VDM-Europe Symposium*, Dublin, Ireland, LNCS
            328, Springer-Verlag, 1988.
[XdR95]   Xu, Q.-W., de Roever, W.-P. and He, J.-F.: Rely–guarantee method for verifying shared
            variable concurrent programs, report 9502. Christian-Albrechts-Universität zu Kiel,
            Germany, 1995.
[XuH91]   Xu, Q.-W. and He, J.-F.: A theory of state-based parallel programming: Part 1. in
            J. Morris and R. Shaw, editors, *Proceedings of BCS FACS 4th Refinement Workshop*
            January 1991, Cambridge, Springer-Verlag.
[Xu92]   Xu, Q.-W.: *A theory of state-based parallel programming*. DPhil. Thesis, Oxford University
            Computing Laboratory, 1992.
[XCC94]   Xu, Q.-W., Cau, A. and Collette, P.: On unifying assumption–commitment style proof
            rules for concurrency. In B. Jonsson and J. Parrow editors, *Proceedings of CONCUR 94*,
            LNCS 836, Springer-Verlag, 1994.