ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**He Jifeng, Oxford University Computing Laboratory, UK, John Cooke,
Loughborough University, UK, and Peter Wallis, University of Bath, UK (Eds)**

BCS-FACS 7th Refinement Workshop

Proceedings of the BCS-FACS 7th Refinement Workshop, Bath, 3-5
July 1996

On Compositionality in Refining Concurrent Systems

Xu Qiwen

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76104-7

On Compositionality in Refining Concurrent Systems

Q. W. Xu

United Nations University

International Institute for Software Technology

P.O. Box 3058, Macau

email: qxu@iist.unu.edu

Abstract

Three different styles of refinement of concurrent systems are investigated. The methods differ in their degree of compositionality. The traditional method considers the refinement of complete systems, and therefore is totally noncompositional. The middle level one is called a modular method, with which one first verifies refinement of each component and then checks that the refinements are compatible by an interference freedom test. The last and more novel one, borrows the rely-guarantee idea from program verification and supports compositional refinement in that one can carry out the development of one process without knowing the structure of other processes. A common example is verified by the various refinement methods. We discuss both advantages and disadvantages of the three approaches, which indicate when it is more suitable to use one particular style.

1 Introduction

Stepwise development is a well-established paradigm for constructing programs, in which a specification is transformed gradually into an implementation. There are two main styles of specification. One is property-oriented, where a list of properties is given; another one is model-oriented, where a specification is given via an abstract program. Formal techniques supporting the two styles are known as program verification and refinement respectively. Since a specification of a practical system is usually very large, often one can focus only on developing one part of the system at a time. This requires the methods to have some compositional properties, so that the pieces developed separately still constitute a correct system.

Components in a concurrent system interact with each other, and the correct functioning of different components is often mutually dependent. Therefore, achieving compositionality in the presence of concurrency is much more difficult than in sequential programming. This issue is extensively studied in program verification [dRHdB⁺94]. The current paper addresses the same problem in refinement.

Three different styles of verification methods with different degrees of compositionality are discussed in [dRHdB⁺94]. They are named *global*, *modular* and *compositional* respectively. In a global method a concurrent system is modelled by a sequential one directly. A modular method (a classic example is the method due to Owicki and Gries [OG76]) typically consists of two steps: firstly, the processes are shown to be locally correct, and secondly, the local proofs are shown to be *interference* free with each other. In a compositional method, a component is developed in a way that the possible interference from its environment is already taken into account, so components are guaranteed to be interference free. We follow exactly the same thread, and discuss the three styles of refinement.

This parallel between verification and refinement is interesting, and actually the refinement rules that we formulate also include verifying some simple properties; corresponding to the styles of the refinement, we use respectively global, modular or compositional rules for establishing such properties.

While the focus of this paper is compositionality, we also address the problem of refining what we call *interface variables*. Interface variables are shared among parallel processes, but are hidden after the parallel composition. Since they do not contribute to the overall observations, it is absolutely possible to replace them

in the next refinement step, but the fact that they are shared among processes has made their refinement more involved in the modular and the compositional approaches.

Our discussion is relevant for many frameworks of refinement, but we need a specific carrier to express our results. We choose the action system formalism of Back and Kurki-Suonio [BKS83] as a paradigm for describing parallel systems. Refinement of action systems was studied by Back, Sere and Wright, in e.g. [BS91, Bac89, BvW94]. Our observations are directly useful to many other transition system based formalisms, e.g., [AL91, KMP94, Jon91].

2 Preliminaries

2.1 Refinement calculus

The basic domains of refinement calculus arise by pointwise extension from the boolean lattice. The truth values

$$\text{Bool} = \{\text{T}, \text{F}\}$$

form a complete lattice under the *implication order*

$$\text{F} \leq \text{T} \quad \text{T} \leq \text{T} \quad \text{F} \leq \text{F}$$

Complement $\neg$, meet $\wedge$ and join $\vee$ are respectively negation, conjunction and disjunction.

Let Σ be a set of states. A *predicate over* Σ is a function $p : \Sigma \rightarrow \text{Bool}$ which assigns a truth value to each state. The set of predicates over Σ

$$\text{Pred}(\Sigma) = \Sigma \rightarrow \text{Bool}$$

also forms a complete lattice under the order obtained from boolean implication by pointwise extension: for $p, q \in \text{Pred}(\Sigma)$

$$p \leq q \quad \text{iff} \quad (\forall \sigma \in \Sigma. p\sigma \leq q\sigma)$$

Complement $\neg$, meet $\wedge$ and join $\vee$ are defined pointwisely too, for example, $(p \wedge q)\sigma \stackrel{\text{def}}{=} (p\sigma \wedge q\sigma)$. The identically false predicate **false** is the bottom, and the identically true predicate **true** is the top, of the predicate lattice. The derived combinator $\Rightarrow$ is defined as usual, that is, $(p \Rightarrow q) \stackrel{\text{def}}{=} (\neg p \vee q)$. In this paper, we assume that binding power of various operators decreases in the order:

$$\begin{array}{l} \text{functional application} \\ \neg \\ \vee, \wedge \\ \Rightarrow \end{array}$$

Pointwise extension of predicates gives us predicate transformers, which are functions of the type

$$\text{Ptran}(\Sigma, \Gamma) \stackrel{\text{def}}{=} \text{Pred}(\Gamma) \rightarrow \text{Pred}(\Sigma)$$

where Σ and Γ are two state spaces. A program statement is identified with the *weakest precondition* predicate transformer [Dij76], which maps a postcondition q to Aq that describes the set of initial states from which A is guaranteed to terminate in states satisfying q . The definition of the predicate transformer regards a program which does not always produce the desired result as bad as one which does not produce the results at all. In particular, a program which cannot guarantee termination is identified with one which is totally nonterminating. For $S, T : \text{Ptran}(\Sigma, \Gamma)$, the *refinement ordering* is

$$S \leq T \stackrel{\text{def}}{=} (\forall q : \text{Pred}(\Gamma). Sq \leq Tq)$$

Under this order, predicate transformers form a lattice: the bottom is **abort**, which maps any postcondition to **false**, and the top is **magic**, which maps any postcondition to **true**; meet and join are again defined pointwisely, for example, $(S \wedge T)q \stackrel{\text{def}}{=} (Sq \wedge Tq)$. Statement **abort** never guarantees anything. Statement **magic** is *miraculous*, for it promises to achieve any postcondition; therefore, it is an imaginary statement, useful only in formal calculations. Meet and join model *demonic* and *angelic* choice respectively.

A statement S is (positively) *conjunctive*, if for any nonempty set of predicates $\{q_i | i \in I\}$,

$$S(\forall i \in I. q_i) \stackrel{\text{def}}{=} (\forall i \in I. Sq_i)$$

We call a conjunctive statement an *action*. The *termination domain* of A is defined by

$$t A \stackrel{\text{def}}{=} A \text{ true}$$

Actions can be combined by any sequential program operators. In particular, a *guarded command* is defined by

$$(b \rightarrow A)q \stackrel{\text{def}}{=} b \Rightarrow Aq$$

and it is quite often useful to denote an action as a collection of alternative sub-actions

$$A_1 \sqparallel \dots \sqparallel A_m \stackrel{\text{def}}{=} A_1 \wedge \dots \wedge A_m$$

2.2 Action systems

When the semantics of a concurrent system is based on *interleaving* observations, it is equivalent to a (non-deterministic) sequential system with respect to logical behaviours: if the system's architecture is completely hidden, there is no way for the observer to tell whether it is run on a sequential, parallel, distributed machine. Therefore, one can use a sequential program to model a concurrent one. An action system $\mathcal{A}$ is a statement of the form

$$\text{local } x; \text{ global } z; \text{ initial } P; \text{ do } A \text{ od}$$

where, as the names suggest, x and z are local and global variables, respectively, and P is the initialisation condition. Statement A is an action; it is executed without interruption once selected; any well-defined sequential commands can be used, as long as the meaning can be given in a predicate transformer calculus.

To allow a concurrent system to be modelled within the action system format, a parallel composition of two action systems must be mapped into another action system. This is straightforward. Two action systems can be composed in parallel if they have the same global but disjoint local variables. Let

$$\mathcal{A}_i \stackrel{\text{def}}{=} \text{local } x_i; \text{ global } z; \text{ initial } P_i; \text{ do } A_i \text{ od}$$

then

$$\mathcal{A}_1 \parallel \mathcal{A}_2 \stackrel{\text{def}}{=} \text{local } x_1, x_2; \text{ global } z; \text{ initial } P_1 \wedge P_2; \text{ do } A_1 \sqparallel A_2 \text{ od}$$

The resulting action system has the same global variables, while its local variables are the union, its initialisation condition is the conjunction, its action the choice, of the respective counterparts in two component systems. Sometimes it is necessary to *hide* some global variables, especially after two systems are composed in parallel. This is indicated by

$$\text{local } l; \mathcal{A}$$

where variables l are made local. We shall later make some minor extensions to the formalism when we discuss compositionality.

2.3 Semantics and refinement

A *computation* is a finite or infinite sequence of states generated by an execution of the action system: the first state satisfies the initial condition, and any state transition is performed by an action. A finite computation is either *terminated* or *aborted*; the former happens when there are no *enabled* actions and the latter occurs when a nonterminating action is enabled. The two kinds of finite computations should be distinguished, and we use a flag to indicate whether the computation is aborted.

A computation *induces* a trace, which is the former with local states and all the *stuttering* transitions (steps that do not change global states) deleted. The terminating/aborting flag is carried over to traces. The semantics of an action system $\mathcal{A}$ is defined as the set of its traces $tr(\mathcal{A})$, which captures the *observable* behaviours of the system.

Action system $\mathcal{C}$ is said to refine action system $\mathcal{A}$, denoted by $\mathcal{A} \sqsubseteq \mathcal{C}$, if for any trace σ of $\mathcal{C}$ either

★ $(\exists \sigma' | \sigma' \in tr(\mathcal{A}). \sigma' \preceq \sigma \wedge aborting(\sigma'))$, or

★ $(\exists \sigma' | \sigma' \in tr(\mathcal{A}). \sigma \preceq \sigma' \wedge nonaborting(\sigma'))$

where $\preceq$ is the prefix relation (including equality). Note that due to these definitions, refinement here only preserves *safety* properties. This is for technical convenience, and the results can be extended to deal with more general properties.

2.4 Proving refinement

Proving refinement essentially amounts to constructing a simulation between the two systems. Simulation is usually divided into so called *forward simulation* and *backward simulation*. In this paper, we consider only forward simulation, which is the most useful one. Similar results hold for backward simulation.

Assume that the local variables of the higher level system are x , the local variables of the lower level system are y , the global variables are z , and the data refinement relation is $R(x, y, z)$. Action A , which refers to variables x and z , is *forward simulated* by action C , which refers to variables y and z , under data refinement relation R

$$A \leq_R C \stackrel{\text{def}}{=} (\forall q. R \wedge Aq \leq C(\exists x. R \wedge q))$$

Simulation of an action system reduces the verification of the complete action systems to proving simulation of individual actions. It presents a way of constructing a higher level computation for a given lower level computation such that the former approximates the latter.

We restrict ourselves to the case that only the lower level system can have extra stuttering actions, which we shall identify in this paper by H or H_i . More precisely, the higher level system $\mathcal{A}$ and lower level system $\mathcal{C}$ that we shall study are basically of the form

$$\begin{aligned} \mathcal{A} &= \text{local } l; (\mathcal{A}_1 \parallel \mathcal{A}_2) \\ \mathcal{C} &= \text{local } k; (\mathcal{C}_1 \parallel \mathcal{C}_2) \end{aligned}$$

where

$$\begin{aligned} \mathcal{A}_i &= \text{local } x_i; \text{ global } z; \text{ initial } P_i; \text{ do } A_i \text{ od} \\ \mathcal{C}_i &= \text{local } y_i; \text{ global } z; \text{ initial } Q_i; \text{ do } C_i \parallel H_i \text{ od} \end{aligned}$$

The following toy example illustrates our major concern. One may wish to refine the action system

<pre> global a, b; initial a = 1; do a > 0 → b := b + 1 [] a < 0 → b := b - 1 od </pre>	into	<pre> global a, b; initial a = 1; do b := b + 1 od </pre>

But when the original action system models one process of a bigger system, whether the above proposed refinement is valid depends obviously on how variable a is used by the other processes. For example, as the environment of the refined process, the action systems

global a, b ; do $a := a + 1$ od	and	global a, b ; do $a := a - 1$ od

lead to two different answers. The same phenomenon arises even when one is refining only local variables in one process (not changing process structure), since the data refinement relation sometimes involves shared variables, and this makes it susceptible to the other processes. The focus of this paper is to investigate refinement of concurrent systems in which such interference can occur.

2.5 A description of the main example

As a running, and more advanced, example, we consider the refinement of mutual exclusion algorithms, studied earlier in temporal logic by Kesten, Manna and Pnueli [KMP94]. Let CS_i be a flag denoting whether or not process i is in its critical section by values 1 or 0 respectively. Mutual exclusion requires that the two processes not be in their critical sections at the same time. Let pc_i^A be the program counter of process i , initially set to 1. Then an abstract mutual exclusion algorithm can be described as follows:

local pc_1^A, pc_2^A ; global CS_1, CS_2 ; initial $pc_1^A = pc_2^A = 1 \wedge CS_1 = CS_2 = 0$; do $A_1^1 : pc_1^A = 1 \rightarrow CS_1 := 0; pc_1^A := 2$ □ $A_1^2 : pc_1^A = 2 \wedge CS_2 \neq 1$ $\rightarrow CS_1 := 1; pc_1^A := 1$ od		do $A_2^1 : pc_2^A = 1 \rightarrow CS_2 := 0; pc_2^A := 2$ □ $A_2^2 : pc_2^A = 2 \wedge CS_1 \neq 1$ $\rightarrow CS_2 := 1; pc_2^A := 1$ od

Statements are labelled for later reference. The two processes are denoted by $\mathcal{A}_1$ and $\mathcal{A}_2$. Since one process can only enter its critical section when the other process is not in a critical section, mutual exclusion property is obviously satisfied. There are two reasons that this abstract algorithm should be refined: first, CS_1 and CS_2 are logical variables used to express specifications and hence are not like program variables (which can be tested, for example); second, one process may have to wait for an arbitrarily long time if the other process is very fast and enters the critical section again before the first process finishes the boolean test. These problems are overcome in a simple algorithm named after Peterson; we shall study how it can be obtained from the abstract algorithm by refinement of different styles.

3 A global approach

The traditional method is global, in that one constructs a complete system as one big action system for the next level and then shows it is a refinement of the original one. The following action system models the well known Peterson's algorithm.

local $pc_1^C, pc_2^C, y_1, y_2, s$; global CS_1, CS_2 ; initial $pc_1^A = pc_2^A = 1 \wedge CS_1 = CS_2 = 0 \wedge s = 1$;

$$\begin{array}{l}
 \text{do} \\
 \quad C_1^1 : pc_1^C = 1 \rightarrow (CS_1, y_1) := (0, \mathbf{F}); pc_1^C := 2 \\
 \quad \square H_1 : pc_1^C = 2 \rightarrow (y_1, s) := (\mathbf{T}, 1); pc_1^C := 3 \\
 \quad \square C_1^2 : pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \\
 \quad \quad \rightarrow CS_1 := 1; pc_1^C := 1 \\
 \text{od}
 \end{array}
 \parallel
 \begin{array}{l}
 \text{do} \\
 \quad C_2^1 : pc_2^C = 1 \rightarrow (CS_2, y_2) := (0, \mathbf{F}); pc_2^C := 2 \\
 \quad \square H_2 : pc_2^C = 2 \rightarrow (y_2, s) := (\mathbf{T}, 2); pc_2^C := 3 \\
 \quad \square C_2^2 : pc_2^C = 3 \wedge (\neg y_1 \vee s = 1) \\
 \quad \quad \rightarrow CS_2 := 1; pc_2^C := 1 \\
 \text{od}
 \end{array}$$

The two processes are denoted by $\mathcal{C}_1$ and $\mathcal{C}_2$. Apart from the new program counters, there are three additional variables y_1 , y_2 and s with which the access of the critical sections are controlled. Process $\mathcal{C}_i$ indicates the wish for entering its critical section by setting y_i to the boolean constant $\mathbf{T}$. Variable s is used to record which process is the latest to request access; when both processes are contesting, the one requested earlier gets the right.

Let the two action systems be

$$\begin{aligned}
 \mathcal{A} &\stackrel{\text{def}}{=} \text{local } x; \text{ global } z; \text{ initial } P; \text{ do } A \text{ od} \\
 \mathcal{C} &\stackrel{\text{def}}{=} \text{local } y; \text{ global } z; \text{ initial } Q; \text{ do } C \square H \text{ od}
 \end{aligned}$$

respectively.

Global method $\mathcal{A} \sqsubseteq \mathcal{C}$, if there exist a refinement relation $R(x, y, z)$ and a predicate $inv(y, z)$ such that

- (i) Initialisation: $Q \leq \exists x. P \wedge R$
- (ii) Main action: $A \leq_{(R, inv)} C$
- (iii) Stuttering action: $\text{skip} \leq_{(R, inv)} H$
- (iv) Invariant: inv is an invariant of $\mathcal{C}$

where $A \leq_{(R, inv)} C$ (and similarly $\text{skip} \leq_{(R, inv)} H$) is defined as

$$\forall q. R \wedge inv \wedge Aq \leq C(\exists x. R \wedge q)$$

This method is actually a minor extension of the traditional method in the refinement calculus literature; the latter is a special case of the former with $inv = \text{true}$. The current rule allows invariants of the lower level system to be separated from the refinement relation. This is sometimes useful in practice, although the extension is not necessary in the global approach. Similar measures become more important in supporting modular and compositional methods. Invariants of the higher level system can also be separated from the refinement relation; details are omitted since they are easy and not needed for our example.

In many practical cases, actions A and C are composed of a number of smaller actions

$$\begin{aligned}
 A &= A_1 \square \dots \square A_m \\
 C &= C_1 \square \dots \square C_m
 \end{aligned}$$

and there is a one-to-one correspondence between the sub-actions; in this case, we can replace proof obligation (ii) in the global refinement method by

$$\bullet A_k \leq_{(R, inv)} C_k \quad \text{for any } 1 \leq k \leq m$$

since this implies the former. Applying the global method to the example, we identify various elements as follows

$$\begin{aligned}
 A &= A_1^1 \square A_1^2 \square A_2^1 \square A_2^2 \\
 C &= C_1^1 \square C_1^2 \square C_2^1 \square C_2^2 \\
 H &= H_1 \square H_2 \\
 d_i &= (pc_i^A = 1 \wedge pc_i^C = 1) \vee (pc_i^A = 2 \wedge pc_i^C = 2) \vee (pc_i^A = 2 \wedge pc_i^C = 3 \wedge y_i) \\
 \bar{g} &= (CS_1 = 1 \Rightarrow y_1 \wedge (\neg y_2 \vee s = 2)) \wedge (CS_2 = 1 \Rightarrow y_2 \wedge (\neg y_1 \vee s = 1)) \\
 &\quad \wedge (s = 1 \vee s = 2) \\
 R &= d_1 \wedge d_2 \wedge \bar{g} \\
 inv &= (pc_1^C = 2 \vee pc_1^C = 3 \Rightarrow CS_1 = 0) \wedge (pc_2^C = 2 \vee pc_2^C = 3 \Rightarrow CS_2 = 0)
 \end{aligned}$$

The proof conditions are straightforward to check.

To complete the global method, we need a rule for proving invariants. This topic is well studied, and the following rule is based on a global approach using notations consistent with the refinement rule.

A global method for proving invariants Predicate $inv(x, z)$ is an invariant of action system $\mathcal{A}$ if there exists a predicate $q(x, z)$ such that

- (i) $P \leq q$
- (ii) $q \wedge t A \leq A q$
- (iii) $q \leq inv$

Remark The advantage of the global method is its simplicity. A concurrent system is directly modelled by a nondeterministic sequential one, and as a matter of fact, parallel composition is not used to express semantics of a specification, but only an indication of a possibly more efficient implementation. One does not need to know anything more than the techniques for refining sequential programs. The disadvantage is that it is totally noncompositional: there is no way to develop the component processes separately; moreover, one often has to handle big predicates, which can be difficult and error-prone.

4 A modular approach

If we look at the proofs of verification conditions of the example in the last section, we can see that for process $\mathcal{C}_i$, the really useful part of the invariant is

$$g_i \stackrel{\text{def}}{=} (CS_j = 1 \Rightarrow y_j \wedge (\neg y_i \vee s = i)) \wedge (s = 1 \vee s = 2) \quad \text{where } i \neq j$$

In fact, letting

$$R_i \stackrel{\text{def}}{=} d_i \wedge g_i$$

we can show easily that process $\mathcal{C}_i$ is a refinement of process $\mathcal{A}_i$ under relation R_i if they are considered as closed systems. This observation can be further exploited to facilitate a modular solution. The question is of course how to ensure that the composed system is also a refinement of the whole original system. The answer is given by Back and von Wright [BvW94], in a simpler setting, where only one process is refined and two parallel processes only share global variables. They suggest adopting from the Owicki/Gries verification method an interference freedom test: refinement holds for the composed systems if each individual refinement relation is not invalidated by execution of the other process.

Notice that variables y_1 , y_2 and s in the lower level system, which are considered local in the global approach, are now shared between the two processes. They are namely interface variables and we now study a modular refinement which also allows the replacement of these variables. In general, a component action system $\mathcal{A}_i$ is of the form

```

local  $x_i$ ;
interface  $l$ ;
global  $z$ ;
initial  $P_i(x_i, l, z)$ ;
do  $A_i$  od

```

Local, interface and global variables form a partition of the variable space. For action systems $\mathcal{A}_1$ and $\mathcal{A}_2$ to be composed in parallel, they must have different local variables, but the same interface and global variables. Interface variables are hidden in the composed system.

Let action system $\mathcal{C}_i$ be

```

local  $y_i$ ;
interface  $k$ ;
global  $z$ ;
initial  $Q_i(y_i, k, z)$ ;
do  $C_i \parallel H_i$  od
    
```

It is obvious that for local refinements to be compatible, the interface variables should be changed consistently. We ensure this by stipulating that both local refinement relations $R_i(x_i, y_i, l, k, z)$ are of the form

$$r_i(x_i, y_i, k, z) \wedge r(l, k, z)$$

in which the first relation does not constrain the higher level interface variables, while the second relation is the same in both R_i and defined over interface and global variables only. We have the following modular method of refinement.

Modular method local l ; $(\mathcal{A}_1 \parallel \mathcal{A}_2) \sqsubseteq$ local k ; $(\mathcal{C}_1 \parallel \mathcal{C}_2)$, if there exist local refinement relations $R_i(x_i, y_i, l, k, z)$ and predicates $inv_i(y_1, y_2, k, z)$ such that

- (i) Local refinement:
  - (1) Initialisation: $Q_i \leq \exists x_i, l. P_i \wedge R_i$
  - (2) Main action: $A_i \leq_{(R_i, inv_i)} C_i$
  - (3) Stuttering action: $\text{skip} \leq_{(R_i, inv_i)} H_i$
- (ii) Invariants: inv_1 and inv_2 are invariants of $\mathcal{C}_1 \parallel \mathcal{C}_2$
- (iii) Interference freedom: for $i \neq j$
  - (1) $R_j \wedge inv_i \wedge tC_i \leq C_i (\forall x_i, l. R_i \Rightarrow R_j)$
  - (2) $R_j \wedge inv_i \wedge tH_i \leq H_i (\forall x_i, l. R_i \Rightarrow R_j)$

One still needs a method for proving invariants. The classical modular method is due to Owicki and Gries [OG76].

A modular method for proving invariants Predicate $inv(x_1, x_2, l, z)$ is an invariant of action system $\mathcal{A}_1 \parallel \mathcal{A}_2$ if there exist two predicates $q_1(x_1, l, z)$ and $q_2(x_2, l, z)$ associated with $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, such that

- (i) Local correctness: for $i = 1, 2$
  - (1) $P_i \leq q_i$
  - (2) $q_i \wedge tA_i \leq A_i q_i$
- (ii) Interference freedom: $q_i \wedge q_j \wedge tA_j \leq A_j q_i$ for $i \neq j$
- (iii) $q_1 \wedge q_2 \leq inv$

We now return to our example. The higher level action system is

<pre> local pc_1^A; global CS_1, CS_2; initial $pc_1^A = 1 \wedge CS_1 = 0$; do $A_1^1 : pc_1^A = 1 \rightarrow CS_1 := 0; pc_1^A := 2$ $\parallel A_1^2 : pc_1^A = 2 \wedge CS_2 \neq 1$ $\rightarrow CS_1 := 1; pc_1^A := 1$ od </pre>	$\parallel$	<pre> local pc_2^A; global CS_1, CS_2; initial $pc_2^A = 1 \wedge CS_2 = 0$; do $A_2^1 : pc_2^A = 1 \rightarrow CS_2 := 0; pc_2^A := 2$ $\parallel A_2^2 : pc_2^A = 2 \wedge CS_1 \neq 1$ $\rightarrow CS_2 := 1; pc_2^A := 1$ od </pre>

And the lower level system is

<pre> local pc_1^C; interface y_1, y_2, s; global CS_1, CS_2; private CS_1, y_1; initial $pc_1^C = 1 \wedge CS_1 = 0 \wedge s = 1$; do $C_1^1 : pc_1^C = 1 \rightarrow (CS_1, y_1) := (0, F); pc_1^C := 2$ $\square H_1 : pc_1^C = 2 \rightarrow (y_1, s) := (T, 1); pc_1^C := 3$ $\square C_1^2 : pc_1^C = 3 \wedge (\neg y_2 \vee s = 2)$ $\rightarrow CS_1 := 1; pc_1^C := 1$ od </pre>	<pre> local pc_2^C; interface y_1, y_2, s; global CS_1, CS_2; private CS_2, y_2; initial $pc_2^C = 1 \wedge CS_2 = 0$; do $C_2^1 : pc_2^C = 1 \rightarrow (CS_2, y_2) := (0, F); pc_2^C := 2$ $\square H_2 : pc_2^C = 2 \rightarrow (y_2, s) := (T, 2); pc_2^C := 3$ $\square C_2^2 : pc_2^C = 3 \wedge (\neg y_1 \vee s = 1)$ $\rightarrow CS_2 := 1; pc_2^C := 1$ od </pre>

Notice that we also declare some variables as private—they are interface or global variables that can only be modified by the process in which they are so identified. Private and local variables can never be changed by other processes. While not necessary, the additional information is helpful to the interference freedom tests. Clearly if a predicate refers only to local and private variables in one process, it will not be invalidated by the other process, so there is no need to perform the concerned interference freedom test. A summary of the various elements in a modular solution for the example is as follows

$$\begin{aligned}
A_i &= A_i^1 \square A_i^2 \\
C_i &= C_i^1 \square C_i^2 \\
&\text{(stuttering actions are as labelled)} \\
d_i &= (pc_i^A = 1 \wedge pc_i^C = 1) \vee (pc_i^A = 2 \wedge pc_i^C = 2) \vee (pc_i^A = 2 \wedge pc_i^C = 3 \wedge y_i) \\
g_i &= (CS_j = 1 \Rightarrow y_j \wedge (\neg y_i \vee s = i)) \wedge (s = 1 \vee s = 2) \quad \text{where } i \neq j \\
R_i &= d_i \wedge g_i \\
inv_i &= (pc_i^C = 2 \vee pc_i^C = 3 \Rightarrow CS_i = 0) \wedge (pc_i^C = 2 \Rightarrow y_i)
\end{aligned}$$

Remark The advantage of the modular approach is that it allows one to concentrate on one process at a time. This advantage disappears if the system is very tightly coupled, because then we do not get much reduction in the complexity of the predicates. However, even for a relatively tightly coupled program, such as a mutual exclusion algorithm, we can still see that the involved predicates are considerably simpler in a modular approach. The modular method is still not compositional in that the interference freedom test is formulated in a way that the complete knowledge of the other process is assumed. After a process is refined locally, we cannot immediately conclude that it is a proper replacement in the complete system; instead we must wait until the other process is developed to check the interference freedom test. Of course one can postpone the interference freedom test and go ahead with another local refinement, but this development work is totally wasted if the interference freedom test turns out to be negative later. This leads us to investigate yet another refinement style, described in the next section.

5 A compositional approach

What matters in interference is not the structure of the environment process, but its relevant effect. In our example for instance, as long as R_i is preserved by environment process $\mathcal{C}_j$, process $\mathcal{C}_i$ functions as a proper replacement of $\mathcal{A}_i$ in the complete system no matter what final structure $\mathcal{C}_j$ has.

Compositional methods are studied extensively in program verification. The principal observation there is that the range of allowed interaction between component processes can often be characterised rather simply (relative to the complete program of the components), and once such interaction is specified, one can then develop a component using the assumed properties that its parallel environment guarantees. Deciding the interface between components is a design step, and like any other design steps, one may make mistakes

or unwise choices. However, in many cases such a step can eliminate a great amount of development effort which would otherwise be wasted.

In shared variable based concurrency, component processes collaborate and at the same time interfere with each other by updating the shared state. To obtain tractable rules for concurrency, the *rely-guarantee* technique was proposed in [Jon83] and further developed e.g. in [Stø91, XH91, Col94] in verification. A *rely-condition* is a predicate over two states, describing the state changes within which that the environment is assumed to stay; a *guarantee-condition* is also a binary state predicate, describing the state changes that the component ensures to satisfy under the given assumption about the environment. By convention, we use primed variables to denote the next state (unprimed variables for the current state); a primed predicate stands for the original predicate with all its variables primed. For our example, we can choose the *rely-condition* of process $\mathcal{C}_i$ as

$$I_i \stackrel{\text{def}}{=} [\text{unchanged}(CS_j, s, y_j) \vee (CS'_j = 1 \Rightarrow y'_j \wedge (\neg y'_i \vee s' = i))] \wedge (s' = s \vee s' = j)$$

where $\text{unchanged}(CS_j, s, y_j) \stackrel{\text{def}}{=} (CS'_j = CS_j \wedge s' = s \wedge y'_j = y_j)$. The *rely-condition* ensures that data refinement relation R_i , which is defined as $d_i \wedge g_i$ in the previous section, will be preserved by the environment of process $\mathcal{C}_i$. To match the assumption of the environment (process $\mathcal{C}_j$ in this case), process $\mathcal{C}_i$ should satisfy a *guarantee-condition*

$$J_i \stackrel{\text{def}}{=} [\text{unchanged}(CS_i, s, y_i) \vee (CS'_i = 1 \Rightarrow y'_i \wedge (\neg y'_j \vee s' = j))] \wedge (s' = s \vee s' = i)$$

We now consider the general case. Let action system $\mathcal{A}$ and $\mathcal{C}$ be

$$\begin{array}{ll} \text{local } x; & \text{local } y; \\ \text{interface } l; & \text{interface } k; \\ \text{global } z; & \text{global } z; \\ \text{private } u; & \text{private } v; \\ \text{initial } P(x, l, z); & \text{initial } Q(y, k, z); \\ \text{do } A \text{ od} & \leq \text{do } C \parallel H \text{ od} \end{array}$$

The *rely-* and *guarantee-conditions* constrain only the shared variables. For action system $\mathcal{C}$ in the following rule, suppose the *rely-* and *guarantee-conditions* are $I(k, z, k', z')$ and $J(k, z, k', z')$. A component cannot insist on its environment interfering, and likewise, the component should be accepted if it does not interfere; therefore, the *rely-* and *guarantee-conditions* are stipulated to be reflexive:

$$k' = k \wedge z' = z \leq I(k, z, k', z') \wedge J(k, z, k', z')$$

Refinement in a parallel context Action system $\mathcal{C}$ refines action system $\mathcal{A}$ in such a parallel context under a refinement relation $R(x, l, y, k, z)$, denoted by $\langle I \rangle \mathcal{A} \leq_R \mathcal{C} \langle J \rangle$, if

(i) Local refinement:

- (1) Initialisation: $Q \leq \exists x, l. P \wedge R$
- (2) Main action: $A \leq_R C$
- (3) Stuttering action: $\text{skip} \leq_R H$

(ii) Interference freedom: $R \wedge I \wedge (x' = x) \wedge (u' = u) \wedge (y' = y) \wedge (v' = v) \leq R'$

(iii) Guarantee condition: $\langle I \rangle \mathcal{C} \langle J \rangle$

The interference freedom condition ensures that the refinement relation will not be invalidated by the environment, where the fact that local and private variables are not changed by the environment is used together with the *rely-condition*. Formula (iii) can be considered as a variant of Hoare triple; it says that any transition from $\mathcal{C}$ satisfies *guarantee-condition* J if any environment transition satisfies *rely-condition* I .

When composed, only processes with matching *rely-* and *guarantee-conditions* can automatically ensure the correct refinement of the complete systems. Obviously, processes may have mutually dependent *rely-* and *guarantee-conditions*, but since both of them describe only safety properties, a simple condition suffices.

Composing refinements local l ; $(\mathcal{A}_1 \parallel \mathcal{A}_2) \sqsubseteq \text{local } k$; $(\mathcal{C}_1 \parallel \mathcal{C}_2)$, if there exist $I_i(k, z, k', z')$, $J_i(k, z, k', z')$ and $R_i(x_i, y_i, l, k, z)$, where $R_i(x_i, y_i, l, k, z) = r_i(x_i, y_i, k, z) \wedge r(l, k, z)$, such that

- (i) (Local) refinement in context: $\langle I_i \rangle \mathcal{A}_i \leq_{R_i} \mathcal{C}_i \langle J_i \rangle$
- (ii) Compatibility: $J_i \leq I_j$ for $i \neq j$

We still need a rule to prove guarantee conditions for a given action system. Let action system $\mathcal{A}$ be the one defined before.

A rule for proving guarantee conditions $\langle I \rangle \mathcal{A} \langle J \rangle$ holds, if there exists a predicate $q(x, l, z)$ such that

- (i) $P \leq q$
- (ii) $q \wedge I \wedge (x' = x) \wedge (u' = u) \leq q[x'/x, l'/l, z'/z]$
- (iii) $q \wedge (l_0 = l) \wedge (z_0 = z) \wedge tA \leq A q \wedge J[l_0/l, z_0/z, l/l', z/z']$

Predicate q is an invariant, because it holds initially (i) and, is preserved by environment actions (ii) as well as action A itself (iii). It is used in (iii) to prove that execution of action A satisfies the guarantee-condition J , where as a standard coding trick, fresh logical variables l_0 and z_0 record the shared state before the transition.

Let us reconsider the example, using the compositional method. We use the following figure to suggest the viewpoint when developing $\mathcal{C}_1$, that is, only the names of the shared variables and the effect of interactions from the other process are known. This should be contrasted with the previous developments where the complete code of the other process is available.

local pc_1^C ; interface y_1, y_2, s ; global CS_1, CS_2 ; private CS_1, y_1 ; initial $pc_1^C = 1 \wedge CS_1 = 0 \wedge s = 1$; do $C_1^1 : pc_1^C = 1 \rightarrow (CS_1, y_1) := (0, F); pc_1^C := 2$ $\square H_1 : pc_1^C = 2 \rightarrow (y_1, s) := (T, 1); pc_1^C := 3$ $\square C_1^2 : pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \rightarrow CS_1 := 1; pc_1^C := 1$ od		interface y_1, y_2, s ; global CS_1, CS_2 ; do $\langle I_2 \quad , \quad J_2 \rangle$ od

The various elements are:

$$\begin{aligned}
 A_i &= A_i^1 \square A_i^2 \\
 C_i &= C_i^1 \square C_i^2 \\
 &\text{(stuttering actions are as labelled)} \\
 d_i &= (pc_i^A = 1 \wedge pc_i^C = 1) \vee (pc_i^A = 2 \wedge pc_i^C = 2) \vee (pc_i^A = 2 \wedge pc_i^C = 3 \wedge y_i) \\
 g_i &= (CS_j = 1 \Rightarrow y_j \wedge (\neg y_i \vee s = i)) \wedge (s = 1 \vee s = 2) \quad \text{where } i \neq j \\
 R_i &= d_i \wedge g_i \\
 I_i &= [\text{unchanged}(CS_j, s, y_j) \vee (CS_j' = 1 \Rightarrow y_j' \wedge (\neg y_i' \vee s' = i))] \wedge (s' = s \vee s' = j) \\
 J_i &= [\text{unchanged}(CS_i, s, y_i) \vee (CS_i' = 1 \Rightarrow y_i' \wedge (\neg y_j' \vee s' = j))] \wedge (s' = s \vee s' = i)
 \end{aligned}$$

Notice that one no longer has the access to the structure of the other process. As a matter of fact, process $\mathcal{C}_1$ can be further developed before process $\mathcal{C}_2$ is constructed. Moreover, any local refinement of the second

process is acceptable for the first process as long as the rely–guarantee pair is respected. The code that we have seen in the last section is one solution, but we can also replace it by

```

local  $pc_2^C$ ;
interface  $y_1, y_2, s$ ;
global  $CS_1, CS_2$ ;
private  $CS_2, y_2$ ;
initial  $pc_2^C = 1 \wedge CS_2 = 0$ ;
do
   $pc_2^C = 1 \rightarrow CS_2 := 0; pc_2^C := 2$ 
   $\square pc_2^C = 2 \rightarrow y_2 := F; pc_1^C := 3$ 
   $\square pc_2^C = 3 \rightarrow (y_2, s) := (T, 2); pc_1^C := 4$ 
   $\square pc_2^C = 4 \wedge (\neg y_1 \vee s = 1) \rightarrow CS_2 := 1; pc_2^C := 1$ 
od

```

(where the first action in the previous solution is split into two now) without disturbing the development of the first process. In general, systems developed with compositional methods are easier to maintain.

Remark The compositional method further delinks the developments of individual processes. It is most suitable when the systems are loosely coupled and becomes ineffective when the interaction between processes is too involved.

6 Discussion

In this paper, we have restricted ourselves to forward simulation. It is known that forward simulation alone is not complete. To deal with situations where forward simulation is insufficient, backward simulation has been proposed (a related method involves introducing so called prophecy variables [AL91]). Our results can also be extended to backward simulation, i.e., there exist modular and compositional rules for backward simulation.

An approach related to our compositional method is proposed in [GNL90], where transition based specifications and rely–guarantee techniques are jointly investigated. But there is one difference: in our approach, there are three elements: an action system and a pair of rely–guarantee conditions, whereas in [GNL90] only the rely–guarantee pair is present (although the guarantee–condition can be expressed by a transition system). Admittedly, our guarantee–condition is somewhat redundant: it can be derived from the action system when the rely–condition is given, so in theory, one can just use the action system as the guarantee–condition. However, the redundancy is retained deliberately, since we wish only to give minimal information through the rely–guarantee conditions when specifying interaction. For our purpose, should we let the action system play also the role of the guarantee–condition, we effectively end up with the modular approach.

Recently Abadi and Lamport [AL93] have studied compositional reasoning of concurrent systems in a TLA setting. They have considered more general properties. Our compositional approach shares some features of their method, but we concentrate on applying rely–guarantee technique to refinement calculus. Abadi and Lamport did not consider the refinement of interface variables.

With the modular method, one can already achieve a considerable amount of compositionality. As a matter of fact, in program verification, in a very theoretical sense it may be said that the modular method contains the compositional one since the completeness proof of the modular verification method [dRHdB⁺94] actually suggests a way to carry out local proofs that are guaranteed to be interference free. Completeness of the modular refinement method has not been investigated, but it is likely that the same results hold also. However, it should be noted that these are mainly of theoretical interest, as in practice, the modular method is effectively used only when other process is also known.

Acknowledgement A large part of the research was carried out within the IRENE-project supported by the Academy of Finland when the author was at Åbo Akademi. I thank Michael Butler for a number of discussions on the modular method and for comments on an earlier version of this paper. Michael in particular suggested a nice soundness proof for the modular method which is included in the appendix. Pierre Collette also provided helpful comments.

References

- [AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
- [AL93] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, SRC Digital Equipment Corporation, 1993.
- [Bac89] R.J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In *REX Workshop for Refinement of Distributed Systems, LNCS 430*, Nijmegen, The Netherlands, 1989. Springer–Verlag.
- [BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142. ACM, 1983.
- [BS91] R.J.R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.
- [BvW94] R.J.R. Back and J. von Wright. Trace refinement of action systems. In B. Josson and J. Parrow, editors, *Proc. 5th International Conference on Concurrency Theory (CONCUR’94)*, LNCS, Uppsala, Sweden, August 1994. Springer–Verlag.
- [Col94] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications - Application to UNITY*. PhD thesis, Université Catholique de Louvain, Belgium, 1994.
- [Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.
- [dRHdB⁺94] W.-P. de Roever, J. Hooman, F. de Boer, Y. Lakhneche, Q.-W. Xu, and P. Pandya. *State-Based Proof Theory of Concurrency: from noncompositional to compositional methods*. Book manuscript, 1994.
- [GNL90] P. Grønning, T.Q. Nielsen, and H.H. Løvengreen. Refinement and composition of transition-based rely–guarantee specifications with auxiliary variables. In K.V. Nori and C.E. Veni Madhavan, editors, *Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS 427*, pages 332–348, Bangalore, India, December 1990. Springer–Verlag.
- [Jon83] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.
- [Jon91] B. Jonsson. Simulations between specifications of distributed systems. In *Proc. 2nd International Conference on Concurrency Theory (CONCUR’91)*, LNCS 527, Amsterdam, The Netherlands, 1991. Springer–Verlag.
- [KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In *Proc. REX Workshop: A Decade of Concurrency, LNCS*, The Netherlands, 1994. Springer–Verlag.

- [OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
- [Stø91] K. Stølen. A method for the development of totally correct shared-state parallel programs. In J.C.M. Baeten and J.F. Groote, editors, *Proc. 2nd International Conference on Concurrency Theory (CONCUR '91)*, LNCS 527, Amsterdam, The Netherlands, 1991. Springer–Verlag.
- [XH91] Q.-W. Xu and J.-F. He. A theory of state-based parallel programming: Part 1. In J. Morris and R. Shaw, editors, *Proc. BCS FACS 4th Refinement Workshop*, pages 326–359, Cambridge, U.K., January 1991. Springer–Verlag.

A Soundness

The soundness of the global method is well established in the literature. The proof consists of showing that if the conditions in the rule are satisfied then the semantical refinement relation holds. We can prove the soundness of the other two rules in the same way, and we indeed have done this for the compositional method. Albeit correct, such proofs are standard. More interestingly, there exists an algebraic proof for the modular method. In this proof, we can show that conditions in the modular method imply the conditions of global refinement, namely, the existence of a modular refinement relation implies the existence of a global refinement relation. Since the global method is sound, the soundness of the modular method follows.

The proof proceeds as follows. Given the conditions in the modular method, the global refinement relation and the global invariant are defined as

$$R \stackrel{\text{def}}{=} R_1 \wedge R_2 \quad \text{inv} \stackrel{\text{def}}{=} \text{inv}_1 \wedge \text{inv}_2$$

The initialisation condition is easy to prove. We concentrate on the condition for actions. The main action of $\mathcal{C}_1 \parallel \mathcal{C}_2$ is $C_1 \parallel C_2$, and we need to prove

$$\forall q. R_1 \wedge R_2 \wedge \text{inv}_1 \wedge \text{inv}_2 \wedge (A_1 \parallel A_2)q \leq (C_1 \parallel C_2)(\exists x_1, x_2, l. R_1 \wedge R_2 \wedge q)$$

By the definition of $\parallel$, it is sufficient to show

$$\forall q. R_1 \wedge R_2 \wedge \text{inv}_i \wedge A_i q \leq C_i(\exists x_1, x_2, l. R_1 \wedge R_2 \wedge q)$$

We first show one simple property: for any assertion q

$$\begin{aligned} & R_i \wedge \text{inv}_i \wedge A_i q \\ \leq & \quad \{\text{local refinement}\} \\ & C_i(\exists x_i, l. R_i \wedge q) \\ \leq & \quad \{C_i \text{ is monotonic}\} \\ & C_i(\exists x_i, l. R_i \wedge q) \wedge C_i \text{true} \end{aligned}$$

Therefore,

$$\begin{aligned} & R_1 \wedge R_2 \wedge \text{inv}_i \wedge A_i q \\ \leq & \quad \{\text{the previous property}\} \\ & C_i(\exists x_i, l. R_i \wedge q) \wedge C_i \text{true} \wedge R_j \wedge \text{inv}_j \\ \leq & \quad \{\text{interference freedom condition}\} \\ & C_i(\exists x_i, l. R_i \wedge q) \wedge C_i(\forall x_i, l. R_i \Rightarrow R_j) \\ \leq & \quad \{C_i \text{ is conjunctive}\} \\ & C_i((\exists x_i, l. R_i \wedge q) \wedge (\forall x_i, l. R_i \Rightarrow R_j)) \\ \leq & \quad \{\text{Predicate Calculus and } C_i \text{ is monotonic}\} \\ & C_i(\exists x_i, l. R_1 \wedge R_2 \wedge q) \\ \leq & \quad \{C_i \text{ is monotonic}\} \\ & C_i(\exists x_1, x_2, l. R_1 \wedge R_2 \wedge q) \end{aligned}$$

The case of the stuttering action is similar (we need in addition a trivial fact: $\text{skip} \parallel \text{skip} = \text{skip}$). □