A Stepwise Development of the Peterson's Mutual Exclusion Algorithm Using B Abstract Systems

J. Christian Attiogbé

LINA - FRE CNRS 2729 - University of Nantes, France
Christian.Attiogbe@univ-nantes.fr

Abstract: We present a stepwise formal development of the Peterson's mutual exclusion algorithm using Event B. We use a bottom-up approach where we introduce the parallel composition of subsystems which are separately specified. First, we specify subsystems as B abstract systems; then we compose the subsystems to get a first abstract solution for the mutual exclusion. This solution is improved to obtain the Peterson's algorithm. This is achieved by refinement and composition of the former abstract subsystems. Therefore the result is formally proved on the basis of correctness (safety) properties added to the invariant. Atelier B (a B prover) is used to check completely the development.

Keywords: Event B, Parallel Composition, Refinement, Mutual Exclusion.

1 Introduction

The B method [1] resides in the category of formal techniques which deal with correct system development starting from (abstract) model-oriented specifications. Stepwise refinement is undertaken until more concrete specifications are obtained or code generated. The refinement steps are formally proved by theorem proving. Consequently, one may build a correct system provided that the initial abstract specification is judiciously captured from the analysis of the informal requirements of the problem at hand. The event-based approach of B [6,2] allows the specification of *abstract systems* which may be used for developing distributed and concurrent systems.
However, the B approach is a top-down approach and its application may be tedious for large system development. In [8] we propose a bottom-up approach to complement the top-down one; our approach provides parallel composition of B abstract systems in order to build large interacting systems by combining their components. This copes well with the practical need to focus on one subsystem at time (without omitting the global identified properties) when developing large systems. Practically it is difficult to formally reason on a very large system; but a solution is to do it through decomposition and reasoning on the subsystems.

Thereby this requires compositionality property in order to have correct system built from the composition of correct and separately built subsystems.

The current paper addresses the development of a system to control the accesses to *critical sections* by processes or subsystems which run concurrently. This topic is a well-studied one; it is known as the mutual exclusion of accesses to critical sections. It is more generally related to the development of a concurrent system from its subsystems. Therefore we concentrate on the illustration of our approach instead of the problem details. The contribution of the paper is a bottom-up technique to build, within Event B, correct interacting systems with access to shared variables.

In the paper we focus on one aspect which concerns global shared variables but our approach of abstract system composition is general; it also deals with message passing (not discussed in the current paper), which is a more general technique in distributed environments. Indeed, techniques of message passing generalise to systems without common memory.

The paper is structured as follows. In Section 2 we present the technique that we use for the parallel composition of B abstract systems. The Section 3 is devoted to the application to the mutual exclusion algorithm: we present a stepwise construction (refinement and composition) of Peterson's algorithm. In Section 4 we discuss some related works and we finish by the Section 5 where we give some concluding remarks.

2 Communicating B Abstract Systems: the Technique

In this section, we begin with the presentation of the working hypothesis and then we present our approach (CBS: Communicating B Systems) through the composition operator introduced to structure abstract systems and to make them communicate. We examine the composition based on the classical communication mechanism of shared state variables.

2.1 Fundamental Preliminaries

Proposition 1. *An abstract system involving several events which cooperate to achieve the same task may be split into several abstract (sub)systems on the basis of the global state variables and the local variables used by these events.*

Variables and Invariant Distribution The variables and the invariant of an abstract system may be distributed over two or more abstract (sub)systems on the basis of the variables used by its events. Some variables are shared by all events, other variables are not. A common part (made of the shared variables and the associated invariant part) of the abstract system is then shared by all events. Accordingly, the remaining variables and invariant may be split to form the desired distribution.

This constitutes a *distribution policy* which is an important working hypothesis in what follows. We define a specific *composition operator* which composes the abstract subsystems in such a way that the result is an abstract system.

Shared Variables and Multiple Substitutions Simultaneous composition of generalized substitutions $(S \| T)$ was initially defined when the generalized substitutions S and T have disjoint space of variables [1]. Here, for the composition of abstract systems, we need the composition of substitutions with non-disjoint space of variables. Therefore, we use the extension of $\|$ proposed by Dunne [18,19]. Several authors have dealt with this concern [14,18,19]. Dunne [18] extends the domain of the multiple composition operator $\|$ and calls it parallel composition of substitutions. He adds the following rule to the initial rewrite rules of Abrial [1]

$$x := E \,||\, x := F \;\; \widehat{=} \;\; E = F \Rightarrow x := E$$

Dunne points out that when the substitutions share the same variable space[1], the generalized composition $\|$ corresponds to the more general[2] *fusion* operator of Back and Butler[11]. Moreover, Dunne's $\|$ parallel composition of substitutions can have an arbitrarily overlapping variable spaces; It is not the case for the fusion operator. The practical issues involved in adopting this approach are: shared variables can be introduced in the specification of abstract systems; concurrent composition is then tractable.

Working Structure of Abstract System We follow the approach presented for abstract machines in [14] by considering the *signature* and the *body* of an abstract system. The signature $signature(\mathcal{S})$ of an abstract system $\mathcal{S}$ is the set of the identifiers appearing in the static part (constants, variables) and in the dynamic part of an abstract system (event names). Consequently, the identifiers are gathered together according to their category (constant, variable, event). A concrete shape of a signature with these features is:

$$\{\langle \texttt{constant}, \{consId\_list\}\rangle, \langle \texttt{variable}, \{varId\_list\}\rangle, \langle \texttt{event}, \{evtId\_list\}\rangle\}$$

where *consId_list*, *varId_list* and *evtId_list* are respectively the list of constant identifiers, the list of variable identifiers and the list of event identifiers.
The signature is required for practical reasons: it is the interface for renaming and comparison of systems. The body $body(\mathcal{S})$ is made of the variables (V), the invariant (Inv), the initialisation (U) and the set of events (E) of the abstract system. We introduce auxiliary functions $sets(\mathcal{S}_i)$, $var(\mathcal{S}_i)$, $inv(\mathcal{S}_i)$, $init(\mathcal{S}_i)$, $events(\mathcal{S}_i)$ to denote respectively the set of sets appearing in the SETS clause, the set of variables, the invariant, the initialisation and the set of events of an abstract system $\mathcal{S}_i$. We simplify the constituents[3] and the notation by considering $\mathcal{S} = \langle \Sigma, B\rangle$ with Σ representing $signature(\mathcal{S})$ and $B = \langle V, Inv, U, E\rangle$ representing $body(\mathcal{S}) = \langle var(\mathcal{S}), inv(\mathcal{S}), init(\mathcal{S}), events(\mathcal{S})\rangle$.

Thus, an abstract system $\mathcal{S}_i = \langle signature(\mathcal{S}_i), body(\mathcal{S}_i)\rangle$ is simply given by $\langle \Sigma_i, B_i\rangle$ or equivalently $\langle \Sigma_i, \langle V_i, Inv_i, U_i, E_i\rangle\rangle$. Moreover, for each event ee mem-

[1] It is called *frame* by Dunne.
[2] It is defined in the general context of all monotonic predicate transformers.
[3] We do not consider all the clauses of an abstract system, however the extension is trivial.

ber of $events(\mathcal{S}_i)$, $guard(ee)$ denotes the guard part of ee and $subst(ee)$ denotes the generalized substitution which describes the action of ee.

Abstract System Renaming A renaming of an abstract system $\mathcal{S} = \langle \Sigma, B \rangle$ is a consistent syntactic replacement of some identifiers used in $\mathcal{S}$ by other given identifiers. Consequently, the renaming is defined on the signature Σ and extended to B. Let Σ_i and Σ_j be signatures (with $\Sigma_i \subseteq \Sigma$) and $\alpha \in \Sigma_i \rightarrow \Sigma_j$ be an injective signature mapping such that the types of the identifiers are preserved; α can be extended easily to B in such a way that each free identifier idt in Σ_i used within B is replaced by its value $\alpha(idt)$ in $\alpha(B)$.

$$\texttt{rename}(\langle \Sigma, B \rangle, \alpha) \mathrel{\widehat{=}} \langle \alpha(\Sigma), \alpha(B) \rangle$$

On this basis and following [14], other auxiliary operations can be defined on abstract systems.

Asynchronous versus Synchronizing Communication Communication involves first the simultaneous evolution of two or more systems, and then the exchange of data. For this purpose, we need composition and communication mechanisms for abstract systems. Above all the involved systems are asynchronous: there is no global clock. From a practical point of view, a simple communication involves a receiver and a sender. Two points of view are generally accepted for communication mechanisms. The communication can be synchronizing (and blocking until completion). This is referred to with the rendez-vous paradigm à la CSP where one considers the final act of communication involving the communicating systems, provided that all the systems reach the communication point. From asynchronous communication point of view, any time duration may pass between the starting of the communication (by one of the involved systems) and its completion (the other involved system participates). It means that systems involved are not blocked before completion. In the scope of the event-driven B approach, events are considered as atomic. Their occurrences are asynchronous, and they do not consume time. They may be synchronizing when the effect of one affects the guard of another one. This is explained in more detail below.

2.2 Composition and Communication with Shared State Variables

A composition operator may permit communication between several abstract systems so as to achieve a common task. We begin with the definition of a composition operator that makes abstract systems communicate through shared state variables. The working hypothesis is that this composition should be compatible with the top-down approach. That means, following on from the result of the composition, it may be possible to use refinement and decomposition[3]. The subsystems to be composed may share some common state variables gv and the associated global invariant properties $I(gv)$. However if the subsystems do not share state variables, the composition results in a pure interleaving. Additionally, each subsystem $\mathcal{S}_i$ may have its own local variables (lv_i). The initialisation operates on gv and lv_i. In the following we use $\mathcal{S}_1$ and $\mathcal{S}_2$ for illustration (Fig.

1). Note that the invariant Inv_i of $\mathcal{S}_i$ is rewritten with local and common state variables of $\mathcal{S}_i$ as: $I_i(gv) \wedge L_i(lv_i) \wedge K_i(gv, lv_i)$.

$L_i(lv_i)$ deals with the local properties, $K_i(gv, lv_i)$ relates local variables and global ones and expresses the associated properties. As already stated, $I_i(gv)$ is the common part of the invariant shared by the abstract systems under consideration. If $K_i(gv, lv_i)$ is not explicit in a given invariant, it is interpreted as the *true* predicate.

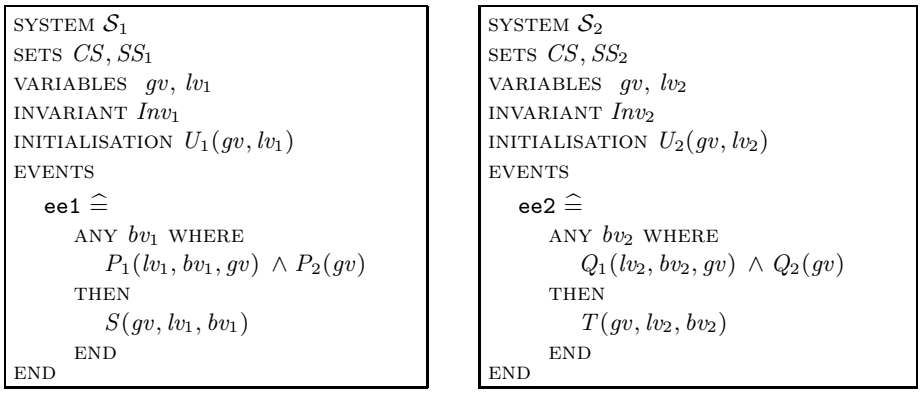

Fig. 1. Abstract systems $\mathcal{S}_1$ and $\mathcal{S}_2$

The shape of events in Figure 1 is used as a canonical form of the event. Each abstract systemmay have several events. The guard *guard*(ee) of an event ee is made of two predicate parts. The first one is expressed using local state variables lv, bound variables bv (variables bound by ANY) and global variables gv of the event: $P_1(lv_1, bv_1, gv)$. The second one is uniquely based on global state variables: $P_2(gv)$. A before-after predicate $BA(v, v')$ is associated to each event and describes it as a predicate relating the values of the state variables before (v) and after (v') the event occurrence.

An event is *enabled* if its guard holds otherwise the event is *disabled*. An event $\mathtt{ee}_i$ enables another event $\mathtt{ee}_j$ if the action of $\mathtt{ee}_i$ contributes in enabling the guard of $\mathtt{ee}_j$. Event guards depend on the state variables. Some events of one of the composed abstract systems may be affected by the action of certain events of the other abstract systems. That means the guard of a particular event may hold after the effect of the other event on the common variables. These events which depend on each other are called *related* events. On the other hand, *unrelated* events are events whose guards do not depend on the actions of the others and vice versa.

Parallel Composition with Asynchronous Communication Asynchronous communication generally refers to the fact that a communication between two or

more subsystems is non-blocking; an arbitrary time duration may pass between the starting of an exchange and its effective completion. It may be contrasted with the rendez-vous paradigm which is blocking and used in the synchronizing case. The asynchronous parallel composition of two abstract systems $\mathcal{S}_1$ and $\mathcal{S}_2$ is denoted by $\mathcal{S}_1]|[\mathcal{S}_2$. This composition defines an abstract system $\mathcal{AS}$ obtained by computing its state and event parts from those of $\mathcal{S}_1$ and $\mathcal{S}_2$. The notation $\mathcal{AS} \mathrel{\widehat{=}} \mathcal{S}_1]|[\mathcal{S}_2$ is then used. A procedure *asynchronousMerging* is used for the computation of the composition result. The procedure is described by the forthcoming inference rules which formalize the computation of each clause of the resulting abstract system.

For the state part, the SETS clause of the resulting abstract system is obtained by merging the SETS clauses of the composed abstract subsystems with a set union: $\{CS, SS_1\} \cup \{CS, SS_2\}$. This is formalized with the **AsyncSetsRule** rule.

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1]|[\mathcal{S}_2 \\ s\mathcal{S}_1 = sets(\mathcal{S}_1) \qquad s\mathcal{S}_2 = sets(\mathcal{S}_2) \qquad s\mathcal{S} = s\mathcal{S}_1 \cup s\mathcal{S}_2\end{array}}{sets(\mathcal{AS}) = s\mathcal{S}}\textbf{AsyncSetsRule}$$

In the same way we formalize using similar inference rules the computation of the other clauses of the composed abstract systems. The VARIABLES clauses of $\mathcal{S}_1$ and $\mathcal{S}_2$ are merged with a set union to form the variables of $\mathcal{S}$: $\{gv, lv_1\} \cup \{gv, lv_2\}$.

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1]|[\mathcal{S}_2 \\ v\mathcal{S}_1 = var(\mathcal{S}_1) \qquad v\mathcal{S}_2 = var(\mathcal{S}_2) \qquad v\mathcal{S} = v\mathcal{S}_1 \cup v\mathcal{S}_2\end{array}}{var(\mathcal{AS}) = v\mathcal{S}}\textbf{AsyncVarsRule}$$

The initialisation of $\mathcal{AS}$ is defined with the merging (parallel composition of substitutions à la Dunne) of the initialisations of $\mathcal{S}_1$ and $\mathcal{S}_2$: $U_1 \| U_2$. We adopt a simplification, the shared variables are not repeated.

The invariant of the resulting $\mathcal{AS}$ abstract system is the conjunction of the $\mathcal{S}_1$ and $\mathcal{S}_2$ invariants: $Inv_1 \wedge Inv_2$ (**AsyncInvRule**). To avoid inconsistency, we require that the invariants of subsystems do not express contradictory requirements. That means one does not imply the negation of the other and vice versa. We note $notContradict(Inv_1, Inv_2)$ for: $K_1(gv, lv_1) \wedge K_2(gv, lv_2)$

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1]|[\mathcal{S}_2 \\ i\mathcal{S}_1 = inv(\mathcal{S}_1) \qquad i\mathcal{S}_2 = inv(\mathcal{S}_2) \qquad notContradict(Inv_1, Inv_2) \\ i\mathcal{S} = i\mathcal{S}_1 \wedge i\mathcal{S}_2\end{array}}{inv(\mathcal{AS}) = i\mathcal{S}}\textbf{AsyncInvRule}$$

The result of this stage of the procedure is presented in the Figure 2 (a).

As far as the event part is concerned, the events of $\mathcal{S}_1]|[\mathcal{S}_2$ are obtained by the union of all the events of the abstract systems $\mathcal{S}_1$ and $\mathcal{S}_2$ (**AsyncEvtRule**). The operator $\biguplus$ denotes such a union of event sets (**MergeEvtRule**).

$$\frac{ee \in events(\mathcal{S}_1) \vee ee \in events(\mathcal{S}_2)}{ee \in events(\mathcal{S}_1) \biguplus events(\mathcal{S}_2)}\textbf{MergeEvtRule}$$

In case of event names conflict, a renaming should be performed before the composition of the abstract systems.

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1]\![\, \mathcal{S}_2 \\ events(\mathcal{S}_1) \cap events(\mathcal{S}_2) = \emptyset \\ e\mathcal{AS} = events(\mathcal{S}_1) \uplus events(\mathcal{S}_2)\end{array}}{events(\mathcal{AS}) = e\mathcal{AS}}\textbf{AsyncEvtRule}$$

The subsystems should be proved to be consistent with respect to their invariant. Therefore, The events of $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) preserve the part of the invariant involving the free variables used in $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) due to variable distribution. The resulting abstract system $\mathcal{AS}$ evolves by one of the observable transition denoted by the events of $\mathcal{S}_1$ or by the events of $\mathcal{S}_2$. The event part of the resulting abstract system has the shape shown in Figure 2 (b). From the observational point of view, the behaviour of $\mathcal{AS}$ is a non-deterministic interleaving of events from $\mathcal{S}_1$ and $\mathcal{S}_2$. Since $\mathcal{S}_1$ and $\mathcal{S}_2$ share global variables, the actions of some events coming from one abstract system may enable (a part of) the guards of some other events coming from the other abstract system. An occurrence of an event is then followed (non-deterministically) by any event (of $\mathcal{S}_1$ or of $\mathcal{S}_2$) whose guard is true. There is a non-deterministic choice if several guards are simultaneously enabled.
To sum up, given $\mathcal{S}_1 = \langle \Sigma_1, \langle\, V_1, Inv_1, U_1, E_1 \rangle\rangle$ and $\mathcal{S}_2 = \langle \Sigma_2, \langle\, V_2, Inv_2, U_2, E_2 \rangle\rangle$, the parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ is defined using the previous rules as follows:

$$\mathcal{S}_1]\![\, \mathcal{S}_2 \mathrel{\widehat{=}} \langle \Sigma_1 \cup \Sigma_2, \langle\, V_1 \cup V_2, Inv_1 \wedge Inv_2, U_1 \| U_2, E_1 \uplus E_2 \rangle\rangle$$

This formalizes the procedure we called *asynchronousMerging* which is the combined use of the rules computing each part.

Algebraic Properties of the Composition

$$\mathcal{S}_1]\![\, \mathcal{S}_2 \equiv \mathcal{S}_2]\![\, \mathcal{S}_1$$

$$(\mathcal{S}_1]\![\, \mathcal{S}_2)]\![\, \mathcal{S}_3 \equiv \mathcal{S}_1]\![\, (\mathcal{S}_2]\![\, \mathcal{S}_3)$$

The parallel composition is commutative and associative. Indeed, firstly the invariant of the composition is the conjunction of the invariants of the components; secondly, the parallel composition of substitutions ($\|$) is used for the initialisation part; finally, for the event part, the composition results in a set of events.

Because of these two properties, the parallel composition of a finite set of abstract systems $\mathcal{S}_i$ is written $]\![\, _{i \in 1 \cdots n} \mathcal{S}_i$. The result is the successive (pairwise) application of $]\![\,$. Therefore the notation is generalized as follows:

$$]\![\, _{i \in 1 \cdots n} \mathcal{S}_i \mathrel{\widehat{=}} \langle \cup_{i \in 1 \cdots n} \Sigma_i, \langle \bigcup_{i \in 1 \cdots n} V_i, \bigwedge_{i \in 1 \cdots n} Inv_i, \|_{i \in 1 \cdots n} U_i, \biguplus_{i \in 1 \cdots n} E_i \rangle\rangle$$

SYSTEM $\mathcal{S}_1]\![[\mathcal{S}_2$

SETS
CS, SS_1, SS_2
VARIABLES
$gv,\ lv_1,\ lv_2$
INVARIANT
$Inv_1 \wedge Inv_2$
INITIALISATION
$U_1 \| U_2$

(a) State Part

SYSTEM $\mathcal{S}_1]\![[\mathcal{S}_2$ (Cont'd)
...
EVENTS

ee1 $\widehat{=}$
ANY bv_1 WHERE
$P_1(lv_1, bv_1, gv) \wedge P_2(gv)$
THEN
$S(gv, lv_1, bv_1)$
END
; ee2 $\widehat{=}$
ANY bv_2 WHERE
$Q_1(lv_2, bv_2, gv) \wedge Q_2(gv)$
THEN
$T(gv, lv_2, bv_2)$
END
END

(b) Event Part

Fig. 2. Abstract system corresponding to $\mathcal{S}_1]\![[\mathcal{S}_2$

We emphasize that our composition approach remains in the initial B framework. We have just temporarily worked on the abstract specification level with composition, nevertheless the B process will continue with refinement (and decomposition).

2.3 Modelling Style with Event B

From a methodological point of view, the presented technique permits a specification style close to the one widely used in the context of process algebra and of action systems. To build larger systems we may specify several subsystems, prove their consistency and compose them gradually. Therefore, this may also help for mastering large systems development. The availability of concurrent communication operators will facilitate the translation from existing related formalisms, based on such operators, into B. Consequently, the B tools may be used following a first specification step based on process algebra and action systems for instance.
Practically, one has to identify interacting subsystems, identify shared resources and specify them in the same way in the subsystems. The interaction between events should then be made explicit (using the composition operator). This is a common working approach already used for programming concurrent processes in operating systems for instance.

3 Development of the Mutual Exclusion Algorithm

We present a development of the well-known bakery algorithm for distributed mutual exclusion. The development is based on the technique introduced in the previous section. In the discussion we use the term "subsystem" instead of "process" as is often encountered in the literature on the topic of mutual exclusion.

3.1 Quick Overview on Mutual Exclusion: Abstract Solution

The bakery's mutual exclusion algorithm was studied in many works[21,25]. In [21] for example, the study is done within the context of temporal logic. We begin with a very abstract version of the mutual exclusion algorithm and then we give a more precise version named Peterson's algorithm following its author's name.

The general problem is that of accesses of the critical sections of code statements, where resources are used or updated by several subsystems. The goal of the algorithm is to avoid simultaneous accesses to these critical sections. In the abstract version one considers two subsystems $\mathcal{P}_1$ and $\mathcal{P}_2$. We specify the B abstract system corresponding to each subsystem. The B specification is quite straightforward. The subsystems to be composed are depicted in the Figure 3.

Each subsystem i has a variable pc_i which indicates if the subsystem is inside (value 0) or outside (value 1) its critical section or interested in entering it (value 2). The accesses to the critical section are protected by the use of Boolean variables cs_1 and cs_2. They respectively indicate that the subsystem $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is within its critical section. Initially the cs_i are set to 0 and the pc_i are set to 1. Each subsystem desiring to enter its critical section sets its cs_i to 0 and additionally its pc_i to 2 (that means its it is ready to enter). A subsystem enters its critical section if it asked for it and if the other subsystem is not within its critical section. In this case the variable cs_i is set properly. On leaving the critical section, the corresponding cs_1 and pc_i are respectively set to 0 and 1.

3.2 The Asynchronous Composition of the Subsystems

The desired system is simply obtained by applying the parallel composition operator to the already defined subsystems (Fig. 3):

$$\mathcal{P}_{12} \mathrel{\widehat{=}} \mathcal{P}_1 ||| \mathcal{P}_2$$

The abstract system resulting from this composition is depicted in Figure 4. It is computed by applying the rules which formalize the procedure *asynchronousMerging* (section 2.2).

From the composition point of view, this first solution works well and illustrates the idea of asynchronous parallel composition we have presented. The main desired property for the system is mutual exclusion:

at most one subsystem is in its critical section at the same time

SYSTEM $\mathcal{P}_1$
VARIABLES
  $cs1, cs2$ /* global variables*/
  $pc1$ /* local variables*/
INVARIANT
  $pc1 \in 0..2$ /* this is $L_1(v_1)$ */
$\wedge$ $cs1 \in 0..1$ /* and now $I_1(gv)$ */
$\wedge$ $cs2 \in 0..1$
INITIALISATION
  $cs1, cs2 := 0, 0 \parallel pc1 := 1$
EVENTS
  `askCS1` $\widehat{=}$
    SELECT $pc1 = 1$
    THEN $cs1 := 0 \parallel pc1 := 2$ END
; `inCS1` $\widehat{=}$
    SELECT $pc1 = 2 \wedge \neg (cs2 = 1)$
    THEN $cs1 := 1$ END
; `outCS1` $\widehat{=}$
    SELECT $(cs1 = 1) \wedge (pc1 = 2)$
    THEN $cs1 := 0 \parallel pc1 := 1$ END
END

SYSTEM $\mathcal{P}_2$
VARIABLES
  $cs1, cs2$ /* global variables*/
  $pc2$ /* local variables*/
INVARIANT
  $pc2 \in 0..2$ /* this is $L_2(v_1)$ */
$\wedge$ $cs1 \in 0..1$ /* and now $I_2(gv)$ */
$\wedge$ $cs2 \in 0..1$
INITIALISATION
  $cs1, cs2 := 0, 0 \parallel pc2 := 1$
EVENTS
  `askCS2` $\widehat{=}$
    SELECT $pc2 = 1$
    THEN $cs2 := 0 \parallel pc2 := 2$ END
; `inCS2` $\widehat{=}$
    SELECT $pc2 = 2 \wedge \neg (cs1 = 1)$
    THEN $cs2 := 1$ END
; `outCS2` $\widehat{=}$
    SELECT $(cs2 = 1) \wedge (pc2 = 2)$
    THEN $cs2 := 0 \parallel pc2 := 1$ END
END

Fig. 3. Concurrent subsystems to be composed

SYSTEM
  $\mathcal{P}_{12}$
VARIABLES
  /* global variables*/
  $cs1$
, $cs2$
  /* local variables*/
, $pc1$
, $pc2$

INVARIANT
  $pc1 \in 0..2$ /* this is $L_1(v_1)$ */
$\wedge$ $pc2 \in 0..2$ /* this is $L_2(v_2)$ */
$\wedge$ $cs1 \in 0..1$ /* and now $I(gv)$ */
$\wedge$ $cs2 \in 0..1$
INITIALISATION
  $cs1, cs2 := 0, 0$
$\parallel$ $pc1 := 1$
$\parallel$ $pc2 := 1$

EVENTS
  `askCS1` $\widehat{=}$
    SELECT $pc1 = 1$
    THEN $cs1 := 0 \parallel pc1 := 2$ END
; `inCS1` $\widehat{=}$
    SELECT $pc1 = 2 \wedge \neg (cs2 = 1)$
    THEN $cs1 := 1$ END
; `outCS1` $\widehat{=}$
    SELECT $(cs1 = 1) \wedge (pc1 = 2)$
    THEN $cs1 := 0 \parallel pc1 := 1$ END
; `askCS2` $\widehat{=}$
    SELECT $pc2 = 1$
    THEN $cs2 := 0 \parallel pc2 := 2$ END
; `inCS2` $\widehat{=}$
    SELECT $pc2 = 2 \wedge \neg (cs1 = 1)$
    THEN $cs2 := 1$ END
; `outCS2` $\widehat{=}$
    SELECT $(cs2 = 1) \wedge (pc2 = 2)$
    THEN $cs2 := 0 \parallel pc2 := 1$ END

END

Fig. 4. Mutually exclusive resulting system

$$\neg\ ((cs1 = 1) \wedge (cs2 = 1))$$

This safety property can be introduced into the invariant of the composed system and completely proved using Atelier B [17].

However, the solution itself is not satisfactory for that it has some drawbacks. For example it is not fair for the composed subsystems; a given event can be observed many times (even infinitely) repeatedly (without allowing other events to occur). We shall go beyond the composition and improve this abstraction by using a refinement. This refinement results in the algorithm of Peterson.

3.3 Refinement: Peterson's Algorithm

To overcome the drawbacks of the previous abstract solution, we consider the Peterson's policy and we show how it can be constructed within the B approach augmented with parallel composition. We also study the correctness of the system by strengthening the invariant after the composition. This leads to correctness proofs that establish the soundness of the solution. Here, the main mechanism to protect accesses to the critical section is based on the use of the (new) Boolean variables y_1 and y_2. They are set to *True* by each subsystem desiring to enter its critical section, and which is ready to do it. Additionally, a variable ss is used to record the number (i) of the process which is the latest to request the access. The other abstract variables $pc1, pc2, cs1, cs2$ are respectively refined by the concrete ones $pc1c, pc2c, cs1c, cs2c$. As with the previous abstract solution, y_1, y_2 and ss are system variables shared by the two subsystems and may be considered as internal to the system. Therefore a subsystem i enters its critical section if its request is not the latest ($ss \neq i$), or the other subsystem (say j) does not request for its critical section: $ss \neq i \vee \neg\ (y_j = True)$.
Consequently, a subsystem i is within its critical section if its variable y_i is set and, the other subsystem is not within its critical section ($\neg\ (y_j = True)$), and if i is not the latest: $((cs_i = 1) \Rightarrow ((y_i = True) \wedge \neg\ (y_j = True) \wedge \neg\ (ss = i)))$.

Refining the previous B abstract subsystems We build the new solution by refining the abstract system $\mathcal{P}_i$. The B refinement process of an abstract system may introduce new variables and new events in the resulting (less) abstract system. The guards of the new system events may be strengthened. The abstract system and its refinement are related by a gluing invariant. The refinement should be proved correct by discharging some proof obligations [5,6,22]: *i)* each introduced new event refines *skip*; *ii)* each abstract event is correctly refined by its corresponding concrete form; *iii)* the introduced new events do not take control for ever; this is achieved by decreasing a variant (included in the refinement) by each occurrence of a new event; *iv)* deadlock-freedom is preserved; considering the disjunction of the event guards.
Consider an abstract system A with variables av and invariant $I(av)$ which is refined by a concrete system C with variables cv and a gluing invariant $J(av, cv)$.

Consider $BAA(av, av')$ and $BAC(cv, cv')$ respectively as the abstract and concrete before-after predicates of the same event, for the correctness of event refinement we have to prove that under the conjunction of the abstract and the concrete invariant, a concrete event (described with $BAC(cv, cv')$) can be simulated ($\exists\, av'$) by an abstract one (described with $BAA(av, av')$) in such a way that the gluing invariant is preserved. Formally

$$I(av) \wedge J(av, cv) \wedge BAC(cv, cv') \Rightarrow \exists\, av'.(BAA(av, av') \wedge J(av', cv'))$$

In the following, we specify an abstract system $\mathcal{P}et_1$ (Fig.5) which refines $\mathcal{P}_1$; this is noted $\mathcal{P}_1 \sqsubseteq \mathcal{P}et_1$. We introduce the new variables $ss, y1, y2$. The invariant of the new system states the type properties of the three new variables. The old variables are retained. The new initialisation trivially establishes the new invariant. We introduce one new event `readyP1`; it refines *skip*. The other events `askCS1`, `inCS1` and `outCS1` are the new specifications of their abstract counterparts. The guard of the event `askCS1` does not change. Its body is updated using the new variables $y1$ and ss following the request policy explained above. The event `inCS1` is refined by changing slightly its guard according to the considered policy. The new guard uses the link between abstract variables and the new concrete ones.
The deadlock-freedom is stated by proving that:
i) the guard of each event ee implies that its substitutions is feasible; it does not establish *False*. (this is proved from $\mathsf{fis}(v := E) = TRUE$ where v is a variable and E is an expression):

$$guard(ee) \Rightarrow \mathsf{fis}(subst(ee))$$

ii) one of the event guards is always true: $(pc1c = 0) \vee (pc1c = 1) \vee ((pc1c = 2) \wedge ((y2 = FALSE) \vee (ss = 2))) \vee (cs1c = 1)$
This is true at the outset because on the initialisation we have $pc1c = 0$; from then and cyclically, the event `readyP1` is enabled. Then it establishes $pc1c = 1$ which in turn is the guard of the event `askCS1`. This one enables the events `inCS1` since $y2$ has not been changed; the body of `outCS1` implies $pc1c = 0$ and the cycle continues.

To sum up, each of the two concurrent subsystems to be composed has the specification (upto a variable renaming) given in the Figure 5. The variables related to $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) are subscripted with 1 (resp. 2).

Composition Now we build the Peterson's algorithm by the composition of the components. First we compose two instances of the subsystem depicted in the Figure 5. Note that the second instance may be obtained by our renaming technique (see Section 2.1). Then, the asynchronous parallel composition is performed in the same way as in the section 3.2.

$$\mathcal{P}eterson\_Alg \;\widehat{=}\; \mathcal{P}et_1 [\![\mathcal{P}et_2$$

The result of the composition is given in the Figure 6.

REFINEMENT $\mathcal{P}et_1$
REFINES $\mathcal{P}_1$
VARIABLES
/* global variables*/
$cs1c, cs2c, ss, y1, y2$
/* local variables*/
$pc1c$
INVARIANT
$pc1c \in 0..2$ /* this is $L_1(v_1)$ */
$\wedge$ $cs1c, cs2c \in 0..1$ /* $I_1(gv)$ */
$\wedge$ $ss \in 1..2$
$\wedge$ $y_1, y_2 \in BOOL$
$\wedge$ $((cs1c = cs1) \vee (cs1c = 0))$
/* glue */
$\wedge$ $((cs2c = cs2) \vee (cs2c = 0))$
$\wedge$ $((pc1c = pc1) \Rightarrow \neg (pc1 = 0))$
INITIALISATION
$cs1c, cs2c, ss := 0, 0, 1$
$\|$ $pc1c := 0$
$\|$ $y1, y2 := FALSE, FALSE$

EVENTS
`readyP1` $\widehat{=}$
SELECT $pc1c = 0$ THEN
$cs1 := 0 \parallel y1 := FALSE$
$\|$ $pc1c := 1$
END
; `askCS1` $\widehat{=}$
SELECT $pc1c = 1$ THEN
$y1 := TRUE \| ss := 1$
$\|$ $pc1c := 2$
END
; `inCS1` $\widehat{=}$
SELECT $(pc1c = 2) \wedge$
$((y2 = FALSE) \vee (ss = 2))$
THEN $cs1c := 1$ END
; `outCS1` $\widehat{=}$
SELECT $(cs1c = 1)$
THEN $cs1c := 0 \parallel pc1c := 0$
END
END

Fig. 5. Refinement of $\mathcal{P}_1$ with Peterson's policy

SYSTEM $Peterson\_Alg$
VARIABLES
/* global variables*/
$cs1c, cs2c, ss, y1, y2$
/* local variables*/
$pc1c, pc2c$
INVARIANT
$pc1c, pc2c \in 0..2 \wedge cs1c, cs2c \in 0..1$
$\wedge$ $ss \in 1..2$
$\wedge$ $y1, y2 \in BOOL$
$\wedge$ $((cs1c = cs1) \vee (cs1c = 0))$
/* glue */
$\wedge$ $((cs2c = cs2) \vee (cs2c = 0))$
$\wedge$ $((pc1c = pc1) \Rightarrow \neg (pc1c = 0))$
$\wedge$ $((pc2c = pc2) \Rightarrow \neg (pc2c = 0))$
INITIALISATION
$cs1c, cs2c, ss := 0, 0, 1$
$\|$ $pc1c, pc2c := 0, 0$
$\|$ $y1, y2 := FALSE, FALSE$

EVENTS
/* they are unchanged */
`readyP1` $\widehat{=}$
$\cdots$
; `askCS1` $\widehat{=}$
$\cdots$
; `inCS1` $\widehat{=}$
$\cdots$
; `outCS1` $\widehat{=}$
$\cdots$
; `readyP2` $\widehat{=}$
$\cdots$
; `askCS2` $\widehat{=}$
$\cdots$
; `inCS2` $\widehat{=}$
$\cdots$
; `outCS2` $\widehat{=}$
$\cdots$
END

Fig. 6. Peterson's algorithm: result of the composition

We may be confident in this result since it is exactly that of the widely known Peterson's algorithm. However, another advantage of our approach is that we can formally state and prove the correctness properties of the obtained algorithm. This is the subject of the following section.

Correctness of the algorithm The mutual exclusion property proved on the abstract version should be maintained: *at most one subsystem is in its critical section at the same time*:

$$\neg\ ((cs1c = 1) \wedge (cs2c = 1))$$

Additionally, we should verify that each subsystem respects the defined conditions when it is within its critical section:

$$(cs1c = 1) \Rightarrow ((y1 = TRUE) \wedge ((y2 = FALSE) \vee (ss = 2)))$$

$$\wedge\ (cs2c = 1) \Rightarrow ((y2 = TRUE) \wedge ((y1 = FALSE) \vee (ss = 1)))$$

Therefore in order to guarantee this within B, we augment the invariant of the abstract system with the conjunction of these properties:

$$\begin{array}{l}(\neg\ (cs1c = 1) \wedge (cs2c = 1)) \\ \wedge\ (cs1c = 1) \Rightarrow ((y1 = TRUE) \wedge ((y2 = FALSE) \vee (ss = 2))) \\ \wedge\ (cs2c = 1) \Rightarrow ((y2 = TRUE) \wedge ((y1 = FALSE) \vee (ss = 1)))\end{array}$$

Monotonicity of the composition We prove that: *the refinement of the composition is the composition of the refinement.*

$$\mathcal{P}_1]\![\,\mathcal{P}_2 \ \sqsubseteq\ \mathcal{P}et_1]\![\,\mathcal{P}et_2$$

This confirms a general result established for our approach. Formally we have

$$\frac{\mathcal{S}_1 \sqsubseteq \mathcal{S}'_1 \qquad \mathcal{S}_2 \sqsubseteq \mathcal{S}'_2}{\mathcal{S}_1]\![\,\mathcal{S}_2 \ \sqsubseteq\ \mathcal{S}'_1]\![\,\mathcal{S}'_2}$$

Consider the context implicitly indicated by the subscripts for each abstract system; by instantiating the refinement proof obligations given above (see 3.3), we have:

$$I_1(av_1) \wedge J_1(av_1, cv_1) \wedge BAC(cv_1, cv'_1) \Rightarrow \exists\, av'_1.(BAA(av_1, av'_1) \wedge J_1(av'_1, cv'_1))$$

$$I_2(av_2) \wedge J_2(av_2, cv_2) \wedge BAC(cv_2, cv'_2) \Rightarrow \exists\, av'_2.(BAA(av_2, av'_2) \wedge J_2(av'_2, cv'_2))$$

The composed systems shared the variables gv only; therefore $av_1 \cap av_2 = gv$. For an event originating from one of the composed systems, it follows from the definition of $]\![$ (conjunction of invariants), the union of events (of composed systems), and that *the shared variables are refined in the same way* in the composed systems, that its concrete description ($BAC(cv_1, cv'_1)$) simulates the abstract one ($BAA(av_1, av'_1)$):

$$I_1(av_1) \wedge I_2(av_2) \wedge J_1(av_1, cv_1) \wedge J_2(av_2, cv_2) \wedge BAC(cv_1, cv_1') \Rightarrow \\ \exists\, av_1'.(BAA(av_1, av_1') \wedge J_1(av_1', cv_1'))$$

This holds for the other events.

Several authors already establish this general result on the monotonicity of composition in various contexts: [10] for the refinement calculus, [16] for action systems, [21,25] for logical frameworks.

Experiment report We use Atelier B to check all the abstract systems and their refinements. The management of shared variables (naming and initialisation) are achieved manually. Note that Atelier B does not manage *abstract systems* directly. But using an encoding into abstract machines, we generate the proof obligations and completely prove the development. As far as this encoding is concerned, the composition is first performed; the new events introduced in refinements are first specified with *skip* in earlier machines; proving the correctness is then straightforward. There is a prototype tool (evt2b) originally developed within the Matisse project [22] which may assist in a systematic translation from B abstract systems into abstract machines.

4 Discussion

Composing systems in a bottom-up manner in B event systems is not a new topic. It has been studied by several authors in the context of process algebra and Action Systems for example. Our approach is therefore very close to the Action Systems view as shown hereafter.

Action Systems View The *Action System* formalism of Back and Kurki-Suonio [12] permits the description of parallel or distributed systems. Actions are guarded statements and are executed atomically. An action A_i has the shape $g_i \rightarrow S_i$ where g_i is the guard and S_i is the statement or the body. The statement can be a non-deterministic choice (noted []) between several other statements S_i. Bottom-up composition has been introduced for action system in [9].

An *action system* enables one to specify the behaviour of a system by a collection of actions. It takes the form:

$$\mathcal{A}_i \;\widehat{=}\; |\,[\,\mathbf{var}\; x_i;\; u_i;\; \mathbf{do}\; A_i\; \mathbf{od}\,]\,|:\; z$$

where x and z are (state) variables; x stands for local variables; z stands for global variables which are used to interact with the environment; u_i stands for the initialisation condition.

The action system formalism provides a parallel composition operator to model concurrent system. The parallel composition of two action systems is achieved if they share some global variables but use disjoint local variables. The composition results in another action system. The latter has the same global variables and the union of the local variables. Its initialisation is the conjunction of the initialisations of the component systems and its action part is made of the choice ([]) of the action part of the component systems.

$$\mathcal{A}_1 \| \mathcal{A}_2 \;\widehat{=}\; |[\, \mathbf{var}\; x_2, x_2;\; u_1 \wedge u_2;\; \mathbf{do}\; A_1\; []\; A_2\; \mathbf{od}\,]| :\; z$$

These composition ideas have also been studied in [16] and within B by Butler et al [15]; they adapt the action systems view (for expressing distributed systems) to the B formalism. Whilst, in [15] an experimental translation from action systems to B machines is given, the specific formal rules for the composition are not given. But these rules are now quite standard for the related formalisms; further developments on these aspects are presented in [22].

We provide a similar composition approach using Event B (with abstract systems instead of abstract machines). But the composition is here completely defined within B. Moreover, the practical advantage of our B approach is the tool availability to assist in the proof steps.

Abrial's Decomposition Approach Abrial is also working on the *decomposition approach* of abstract systems to split a large system into smaller ones using shared variables [4]. The ideas are similar in that the global system is a composition of several interacting subsystems. However he deals with the top-down approach going from the global system to the subsystems. Moreover there is not an explicit composition operator. In our bottom-up approach, the shared variables and the associated invariant are effectively elements of the top level. The shared variables should be refined in the same manner; this is also a constraint of the decomposition approach. We have an explicit composition operator (à la Process Algebra) expandable to explicit message passing. Therefore the approaches are not orthogonal, they are complementary.

As far as the B method is concerned, there are some works related to composition and interaction between specifications. An example is the work by Schneider and Treharne [24,23] on composing CSP and B. CSP processes [20] are used to describe controllers for B machines. The controllers handle the control flow of machine operations without sharing machine states. The machine model and the controller model are developed separately. The composition of B machines is done here through the CSP controllers of the involved B machines. This approach is highly CSP-driven even if the machines part may be developed within the B framework. In our approach B systems are used and the control part is incorporated in the event guards.

The *Assumption-Commitment* approach (also called *Rely-Guarantee*) [27,26] has been proposed for the composition of concurrent systems with shared state variables. Briefly, it consists for each system involved in the composition, to establish correctness properties by making some assumptions about the other systems which constitute its environment. Therefore, the design of the component systems are not really independent and this makes the structuring of specifications tedious. The *Assumption-Commitment* approach does not permit independent refinement. Our composition approach does not constrain the composed systems to reason about their environment. The components are inde-

pendent but the correctness properties are treated with proof obligations during the composition. This simplifies the structuring of the global system and also the independent refinement of the subsystems. Indeed, the interference between global variables are considered only during the composition.

5 Concluding Remarks

In this paper, we presented a complete development of a concurrent system by combining composition techniques, refinement and tools. First, a composition approach (bottom-up) to build interacting concurrent systems within Event B is presented. Then it is used for a development: the construction of Peterson's mutual exclusion algorithm by refinement from an earlier abstract version. Currently, we use global variables to ensure the communication between the interacting subsystems. Atelier B is used to prove the complete development. Only safety properties are considered here; but we have investigated liveness properties in [7] for a subset of Event B, by combining B and the Spin Model checker.

As far as composition is concerned, in [16] Butler deals with the refinement of communicating action systems. We share some features with his work: a compositional approach. But the main difference is that Butler's approach is based on communication with shared events (occurrences of system transitions which are commonly named) instead of shared variables as we presented here.

The contribution of our work can be underlined through several points. First, the systematic construction of software systems using well-defined techniques: composition (bottom-up approach), refinement and theorem proving. Second, the effective use of available tools to support this construction.

Some aspects of the presented work are the subject of ongoing development; for example some dedicated interfaces in front of the B tools (including evt2b), and the development of many other real size case studies to assess and improve the proposed approach. Other communication operators are needed. Yet we have experimented a technique for message passing; we use specific variables to handle messages; but more work on this technique of message passing is necessary, to make the development of large distributed systems practical. Besides, we are working on a procedure to translate from process algebra specifications into Event B systems.

Acknowledgments: Many thanks to my colleagues and to the anonymous referees for their valuable comments on the current work.

References

1. J-R. Abrial. *The B Book*. Cambridge University Press, 1996.
2. J-R. Abrial. Extending B without Changing it (for developping distributed systems). *Proc. of the 1st Conference on the B method, H. Habrias (editor), France*, pages 169–190, 1996.

3. J-R. Abrial. Event Driven Distributed Program Construction. MATISSE project, August 2001.
4. J-R. Abrial. Discrete System Models. Internal Notes (available at www-lsr.imag.fr/B, B Working Group), February 2002.
5. J-R. Abrial, D. Cansell, and D. Mery. Formal Derivation of Spanning Trees Algorithms. In D. Bert et al., editor, *ZB'2003 – Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 457–476. Springer-Verlag, 2003.
6. J-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *Proc. of the 2nd Conference on the B method, D. Bert (editor)*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer-Verlag, 1998.
7. C. Attiogbé. A Mechanically Proved Development Combining B Abstract Systems and Spin. In *Proceedings of the 4th International Conference on Quality Software (QSIC 2004)*. IEEE Computer Society Press.
8. C. Attiogbé. Communicating B Abstract Systems (CBS). Technical Report 02.08, IRIN, University of Nantes, December 2002.
9. R. J. Back and K. Sere. From Action Systems to Modular Systems. *Software - Concepts and Tools*, 17(1):26–39, 1996.
10. R-J. Back and J V Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
11. R. J. R. Back and M. J. Butler. Fusion and Simultaneous Execution in the Refinement Calculus. *Acta Informatica*, 35(11):921–949, 1998.
12. R.J. Back and R. Kurki-Suonio. Decentralisation of Process Nets with Centralised Control. In *Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM, 1983.
13. D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors. *ZB'2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, France*, volume 2272 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
14. D. Bert, M-L. Potet, and Y. Rouzaud. A Study on Components and Assembly Primitives in B. *Proc. of the 1st Conference on the B method, H. Habrias (editor), France*, pages 47–62, November 1996.
15. M. Butler and M. Walden. Distributed System Development in B. *Proc. of the 1st Conference on the B method, H. Habrias (editor), France*, pages 155–168, 1996.
16. M. J. Butler. Stepwise Refinement of Communicating Systems. *Science of Computer Programming*, 27(2):139–173, 1996.
17. ClearSy. *Atelier B V3.6*. Steria, Aix-en-Provence, France.
18. S. Dunne. The Safe Machine: A New Specification Construct for B. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 472–489, 1999.
19. S. Dunne. A Theory of Generalised Substitutions. In Bert et al. [13], pages 270–290.
20. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, NJ, 1985.
21. Y. Kesten, Z. Manna, and A. Pnueli. Temporal Verification of Simulation and Refinement. In *REX Symposium A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 273–346. Springer-Verlag, 1994.
22. MATISSE. Handbook for Correct Systems Construction. Technical Report IST-1999-11345, EU-Project MATISSE: Methodologies and Technoloies for Industrial Strength Systems Engineering,University of Southampton, April 2003.
23. S. Schneider and H. Treharne. Verifying Controlled Components. In E. Boiten, J. Derrick, and G. Smith, editors, *Proc. of the Integrated Formal Methods (IFM'2004)*, volume 2999 of *Lecture Notes in Computer Science*, pages 87–107. Springer-Verlag, 2004.

24. S. Schneider and H. Treharne. Communicating B Machines. In Bert et al. [13], pages 416–435.
25. Q. Xu. On Compositionality in Refining Concurrent Systems. In J. He, J. Cooke, and P. Wallis, editor, *Proceedings of the BCS FACS 7th Refinement Workshop*. Springer-Verlag, 1996.
26. Q. Xu, W. P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.
27. Q. Xu and M. Swarup. Compositional Reasoning Using the Assumption-Commitment Paradigm. *Lecture Notes in Computer Science*, 1536:565–583, 1998.