`Will be presented at Concur'94, Uppsala, Sweden, August 22-25, 1994.`

On Unifying Assumption–Commitment Style Proof Rules for Concurrency

Qiwen Xu[1]*, Antonio Cau[2]** and Pierre Collette[3]***

[1] Department of Computer Science, Åbo Akademi, Lemminkainenkatu 14, SF-20520 Turku, Finland,

[2] Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Preußerstr. 1-9, D-24105 Kiel, Germany,

[3] Unité d'Informatique, Université Catholique de Louvain, Place Sainte-Barbe 2, B-1348 Louvain-la-Neuve, Belgium

Abstract. Assumption–Commitment paradigms for specification and verification of concurrent programs have been proposed in the past. We show that two typical parallel composition rules for shared variable and message passing programs [8, 12] which hitherto required different formulations are instances of one general rule mainly inspired by Abadi & Lamport's composition theorem [1].

1 Introduction

Compositional methods support the verify-while-develop paradigm (an interesting account is given in [15]). However, compared to sequential programs, concurrent programs are much harder to specify and verify. In order to obtain tractable proof rules for concurrency, *assumption–commitment* (sometimes also called *rely–guarantee*), as against monolithic, specification paradigms have been proposed, in which a component is verified to satisfy a commitment under the condition that the environment satisfies an assumption. Such proof systems have been studied for concurrent programs communicating through shared variables [8, 17–19], as well as through message passing (as in `OCCAM` for example) [12, 14, 20]. Although historically these two systems were developed independently, it has been noticed from the beginning that the proof rules (recalled in Sect. 3) look remarkably similar. Nevertheless, there is a puzzling difference. Indeed, suppose the processes P_1 and P_2 satisfy the assumption-commitment pairs A_1–C_1 and A_2–C_2 respectively. Then, in the case of shared variable concurrency, $P_1 \| P_2$ satisfies the assumption-commitment pair A–C if the following premises hold:

$$A \vee C_1 \rightarrow A_2 \qquad A \vee C_2 \rightarrow A_1 \qquad C_1 \vee C_2 \rightarrow C$$

* Partially supported by ESPRIT Project 6021 (REACT) and Finnish Akademi Project IRENE.

** Partially supported by ESPRIT Project 6021 (REACT).

*** Supported by National Fund for Scientific Research (Belgium).

But in the case of message passing concurrency, the corresponding premises are:

$$A \wedge C_1 \rightarrow A_2 \qquad A \wedge C_2 \rightarrow A_1 \qquad C_1 \wedge C_2 \rightarrow C$$

In this paper, we take the convention that operators $\vee$ and $\wedge$ bind more closely than $\rightarrow$ (and than $\leftrightarrow$, used in latter sections). Intuitively, the use of disjunction for shared variable programs and conjunction for message passing programs can be understood from the following observations:

– In [8, 17–19], assumptions and commitments are constraints on atomic steps; a step of $P_1 \| P_2$ is either a step of P_1 *or* a step of P_2.
– In [12, 14, 20], assumptions and commitments are predicates on communication traces; a trace of $P_1 \| P_2$ is both a trace of P_1 *and* a trace of P_2.

This topic has not caught too much attention, largely because the two worlds of concurrency did not meet until recently the effort of writing a comprehensive book on verification of concurrent programs [16] which contains both systems has been undertaken. For this effort it becomes desirable to build a connection between the two rules, especially to obtain a rule which applies to *mixtures* of shared variable and communication based concurrency.

In the meantime, there has been a number of efforts on general methods for composing assumption-commitment specifications, such as Abadi & Lamport's composition theorem [1] and its reformulation in [6] at the semantical level, and the systems in [10, 5, 2] at the proof theoretic level. Abadi & Lamport's composition theorem is particularly powerful, and it is speculated in [1] that it can be applied to several verification methods.

In this paper, we show that the proof rules for the two styles of concurrency are special instances of a new general rule for parallel composition; the latter is based on Abadi & Lamport's composition theorem. In Sect. 2 we first illustrate with two examples the semantics for both shared variable and message passing concurrency. Section 3 presents the two specific assumption–commitment rules for the two types of concurrency. Section 4 gives the general assumption-commitment parallel composition rule. In Sect. 5 we prove that the specific rules are instances of the general rule. We conclude the paper with a short discussion in Sect. 6.

2 Computations

A concurrent process is by definition run together with some other processes. When we consider one process in particular, we call it component, and call the other processes its environment. The execution of a process is modeled by interleaving its atomic actions with those from the environment. A behavior is a sequence

$$\sigma : \ \sigma_0 \overset{a_1}{\rightarrow} \sigma_1 \overset{a_2}{\rightarrow} \sigma_2 \overset{a_3}{\rightarrow} \cdots$$

where each σ_i is a state, a_i denotes an agent, and the sequence is either infinite or ends in a state σ_m for some $m \geq 0$. The length $|\sigma|$ of a behavior σ is the number of transitions in it. For any finite k, such that $0 \leq k \leq |\sigma|$, $\sigma|_k$ is the prefix of σ of length k and σ_k is the $(k+1)$-th state of σ.

Agents are used to distinguish actions performed by the environment from those performed by the component. Consequently, in principle two agents suffice: the environment and the component in question. However, allowing arbitrary sets of agents eases the composition problem: in the compositional approach, semantic parallel composition is basically the set-intersection of behaviors of its composing processes [1]. Proof theoretically, a parallel composition rule can be formulated with logical conjunction which corresponds to set intersection.

In this paper, we do not wish to adopt a specific semantics. However, to illustrate that set intersection, as semantical operator, corresponds to parallel composition, we outline a possible semantics with help of simple examples.

Consider the concurrent program $x := x + 1 \parallel x := x + 2$ first. As one of the many possibilities, we choose $\{a\}$ and $\{b\}$ as the agent sets for the first and second process respectively. All other agents, e.g. $\{c, d\}$ are considered to be agents from the outside environment. A state η is a mapping from program variables to values; $\eta\surd$ indicates that this is a terminated state, in which the symbol "$\surd$" is pronounced as "tick", adopting an early convention introduced by Hoare. Some typical behaviors from the viewpoints of $x := x + 1$ and $x := x + 2$ are respectively:

$$\begin{array}{l|l}
0 \xrightarrow{c} 5 \xrightarrow{a} 6\surd & 0 \xrightarrow{d} 5 \xrightarrow{a} 6 \\
0 \xrightarrow{d} 5 \xrightarrow{a} 6 & 0 \xrightarrow{b} 2 \\
0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{b} 8 & 0 \xrightarrow{b} 2\surd \\
0 \xrightarrow{c} 5 \xrightarrow{a} 6\surd \xrightarrow{b} 103\surd & 0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{b} 8 \\
0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{b} 8\surd & 0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{b} 8\surd
\end{array}$$

To support compositionality when defining the semantics of one process, the information of the other process should not be used; b-transitions are thus arbitrary from the point of view of $x := x + 1$. Note also our treatment of the termination flag $\surd$: it is allowed to be raised only if the component has finished executing its code, but it can also be raised later when an arbitrary number of subsequent environment transitions have been performed.

Only the matching behaviors of $x := x + 1$ and $x := x + 2$ are retained in the semantics of their parallel composition. Thus, typical behaviors of the program $x := x + 1 \parallel x := x + 2$ are:

$$\begin{array}{l}
0 \xrightarrow{d} 5 \xrightarrow{a} 6 \\
0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{b} 8 \\
0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{b} 8\surd
\end{array}$$

When considering the effect of $x := x + 1 \parallel x := x + 2$ as a whole program, it should not matter which process is responsible for which state transition. In general, $[\![\mu : P]\!]$ is the set of behaviors in which transitions from P are labeled by *any* agent in the set μ and transitions from the environment are labeled by *any* agent in the set $\overline{\mu}$. Formally speaking, let $\sigma \simeq_\mu \sigma'$ denote that one behavior can be derived from the other by changing some occurrence of μ–agents by other μ–agents and some occurrence of $\overline{\mu}$–agents by other $\overline{\mu}$–agents. Then, $[\![\mu : P]\!]$ is closed with respect to $\simeq_\mu$ (μ-abstraction [1]). Returning to the above example, other behaviors of the parallel composition are:

$$
\begin{array}{l}
0 \xrightarrow{d} 5 \xrightarrow{b} 6 \\
0 \xrightarrow{c} 5 \xrightarrow{a} 6 \xrightarrow{a} 8 \\
0 \xrightarrow{c} 5 \xrightarrow{b} 6 \xrightarrow{b} 8 \\
0 \xrightarrow{d} 5 \xrightarrow{a} 6 \xrightarrow{a} 8\surd \\
0 \xrightarrow{d} 5 \xrightarrow{b} 6 \xrightarrow{b} 8\surd
\end{array}
$$

Let $S^{\simeq_\mu}$ denote the closure of a set of behaviors S under μ-abstraction. Then, for any choice of μ, μ_1, and μ_2 such that $\mu = \mu_1 \cup \mu_2$ and $\mu_1 \cap \mu_2 = \emptyset$, the semantics of parallel composition is defined as

$$[\![\mu : P_1 \parallel P_2]\!] = (\,[\![\mu_1 : P_1]\!] \cap [\![\mu_2 : P_2]\!]\,)^{\simeq_\mu}$$

We now consider `OCCAM`-like programs. In `OCCAM`, only local variables are allowed and processes communicate by passing messages over named channels. In a semantics, communication traces, which are sequences of records containing channel names and communicated values, are included. Therefore, a state now maps program variables to values and a special variable h (called history variable) to communication traces. Consider the program

$$P_1 \stackrel{\text{def}}{=} a?x; b!x^2; c?y \qquad \parallel \qquad P_2 \stackrel{\text{def}}{=} b?z; c!z^3$$

It consists of two processes: the first process receives a number, calculates its square, then passes the result to the second process; after receiving a value, the second process calculates its cube and sends the result back to the first process. As one of the possibilities, we simply use process names P_1 and P_2 as the agents when the transition is an *output* communication from that process (internal computation steps, which are not present in this simple example, can be labeled in the same way). Assume the overall environment process has the name P_e. Without being verbose as in the previous example, we only give one interesting behavior of their parallel composition, in which the variables are ordered as (h, x, y, z) and $<>$ stands for the empty trace:

$$
\begin{array}{l}
(<>, 0, 0, 0) \xrightarrow{P_e} (<(a,3)>, 3, 0, 0) \xrightarrow{P_1} (<(a,3)(b,9)>, 3, 0, 9) \\
\xrightarrow{P_2} (<(a,3)(b,9)(c,729)>, 3, 729, 9)\surd
\end{array}
$$

3 Specific Assumption–Commitment Proof Rules

In this section, we review the specific assumption–commitment proof rules for the parallel composition of respectively shared variable and `OCCAM`-like programs. We only consider verification of weak total correctness, i.e., partial correctness plus the fact that the program does not perform an infinite number of actions. Total correctness for concurrent programs additionally requires that no deadlock occurs; however, coping with deadlock-freedom requires a more sophisticated model [7] than the one presented in Sect. 2.

Specifications are tuples $(p,\ A,\ C,\ q)$, in which state predicates p and q are respectively pre and post conditions while, in order to achieve compositionality, the A–C pair describes the interaction between the component and the environment. In shared variable concurrency, programs interact by interleaved atomic updates of a shared state, and therefore, the A–C pair describes the possible state changes by the environment and the component respectively; in message passing concurrency, programs interact by joint communications, and thus the A–C pair describes communication traces.

The informal interpretation for a process P to satisfy a specification tuple $(p,\ A,\ C,\ q)$ is as follows: for any behavior σ of P

I) if precondition p holds initially in σ, and A holds over any prefix of σ, then C holds after the transition extending that prefix,
II) if precondition p holds initially in σ and A holds over σ, then
  - P performs only a finite amount of transitions,
  - if σ is a terminated behavior, then the final state satisfies postcondition q

3.1 Shared Variable Programs

In this setting, A and C are binary state predicates. The convention is to use unprimed variables to refer to the state before and primed variables to refer to the state after the transition. For example, $(\eta, \eta') \models x' \geq x$ if the value of x in η' is greater than or equal to the value of x in η. For any behavior σ, define

⋆ A holds over σ iff $(\sigma_i, \sigma_{i+1}) \models A$ for any environment transition (σ_i, σ_{i+1}) in σ

and

⋆ C holds over σ iff $(\sigma_i, \sigma_{i+1}) \models C$ for any component transition (σ_i, σ_{i+1}) in σ

The parallel composition rule for shared variable programs is:

$$\frac{\begin{array}{lll} P_1\ \underline{sat}\ (p,\ A_1,\ C_1,\ q_1) & A \vee C_1 \rightarrow A_2 & \\ P_2\ \underline{sat}\ (p,\ A_2,\ C_2,\ q_2) & A \vee C_2 \rightarrow A_1 & C_1 \vee C_2 \rightarrow C \end{array}}{P_1 \parallel P_2\ \underline{sat}\ (p,\ A,\ C,\ q_1 \wedge q_2)}$$

The assumption A_1 specifies the state changes that the component process P_1 can tolerate from its environment. Both state changes by process P_2 (for which C_2 is guaranteed) as well as state changes of the overall environment (for which A is assumed) must be viewed as state changes by the environment of P_1. Since those state changes are *interleaved* in the execution model (see Sect. 2), the condition

$A \vee C_2 \rightarrow A_1$ is precisely the one needed to ensure that the assumption of process P_1 is respected by its environment. Similar arguments also hold for process P_2. Finally, if both C_1 and C_2 are guaranteed, then the condition $C_1 \vee C_2 \rightarrow C$ ensures that C is guaranteed for the state changes by $P_1 \| P_2$

Example: Consider the problem [9, 13] of finding the index of the first positive number in an integer array $X(1, \ldots, n)$. The following program consists of two parallel processes, one searching the odd and the other the even positions. A global variable k is used to store the index found. Initially, k is set to $n+1$ and is not changed if there are no positive numbers in the array. Two local search pointers i and j are used. At each position, a process checks whether its search pointer is still below k (otherwise it terminates) and then a process advances the pointer or sets k to the value of the pointer depending on the current value of the array at that position.

```
$i := 1;$
do $\langle i < k \wedge X[i] > 0 \rightarrow k := i \rangle \;\square\; \langle i < k \wedge X[i] \leq 0 \rightarrow i := i + 2 \rangle$ od

$j := 2;$
do $\langle j < k \wedge X[j] > 0 \rightarrow k := j \rangle \;\square\; \langle j < k \wedge X[j] \leq 0 \rightarrow j := j + 2 \rangle$ od
```

The brackets "$\langle$" and "$\rangle$" indicate that the alternatives separated by "$\square$" are executed atomically. The first process satisfies the specification:

$p: \quad k = n + 1$
$A_1: (k' = k \vee (k' < k \wedge X(k') > 0)) \wedge X' = X \wedge i' = i$
$C_1: (k' = k \vee (k' < k \wedge X(k') > 0)) \wedge X' = X \wedge j' = j$
$q_1: \quad (X(k) > 0 \vee k = n + 1) \wedge (\forall u.\ odd(u) \wedge X(u) > 0 \rightarrow k \leq u)$

Condition A_1 expresses that the environment does not change the array, nor the local search pointer i; moreover, if k is changed, it can only be decreased and the new index corresponds to a positive number in the array. Condition C_1 describes what the component guarantees, matching the assumption of the second process. Post condition q_1 says that the final value of k is either $n+1$ or points to a positive number in the array, and that if there are positive numbers at odd positions then the final value of k is smaller than or equal to the smallest index of them.

Similarly, the second process satisfies the specification:

$p: \quad k = n + 1$
$A_2: (k' = k \vee (k' < k \wedge X(k') > 0)) \wedge X' = X \wedge j' = j$
$C_2: (k' = k \vee (k' < k \wedge X(k') > 0)) \wedge X' = X \wedge i' = i$
$q_2: \quad (X(k) > 0 \vee k = n + 1) \wedge (\forall v.\ even(v) \wedge X(v) > 0 \rightarrow k \leq v)$

With the parallel composition rule for shared variables, we can infer that the parallel composition of both processes satisfies:

$p: \; k = n + 1$
$A: k' = k \wedge X' = X \wedge i' = i \wedge j' = j$
$C: true$
$q: \quad (X(k) > 0 \vee k = n + 1) \wedge (\forall w.\ X(w) > 0 \rightarrow k \leq w)$

The postcondition indicates that k has the index of the first positive number if there exists one.

3.2 `OCCAM`-like Programs

In this setting, A and C are predicates over the history variable h. In any state of a behavior, h stores all the communications performed so far. The projections of the history onto channels a, and both a and b for example are denoted by h_a and h_{ab} respectively. For any behavior σ, define

$\star$ A holds over σ iff $\sigma_i \models A$ for any state σ_i in σ

and

$\star$ C holds over σ iff $\sigma_i \models C$ for any state σ_i in σ

When restricting to an `OCCAM`-like model, one has the following parallel composition rule [14, 20]:

$$\frac{\begin{array}{lll} P_1\ \underline{sat}\ (p,\ A_1,\ C_1,\ q_1) & A \wedge C_1 \rightarrow A_2 & p \wedge A \rightarrow A_1 \wedge A_2 \\ P_2\ \underline{sat}\ (p,\ A_2,\ C_2,\ q_2) & A \wedge C_2 \rightarrow A_1 & C_1 \wedge C_2 \rightarrow C \end{array}}{P_1 \parallel P_2\ \underline{sat}\ (p,\ A,\ C,\ q_1 \wedge q_2)}$$

Suppose process P_1 has two input channels a and b, with channel a from the overall environment and channel b from P_2. Assume (by A) that the overall environment either sends nothing or number 2 on channel a, and suppose process P_2 guarantees (by C_2) to send nothing or number 5 on channel b. Then the combined information process P_1 may assume is the *conjunction* of the two: the only possible message on channel a is 2 and the only possible message on channel b is 5. This motivates the condition $A \wedge C_2 \rightarrow A_1$.

Example: Consider the simple program $P_1 \overset{\text{def}}{=} a?x; b!x^2; c?y \parallel P_2 \overset{\text{def}}{=} b?z; c!z^3$ again. Assume the input value on channel a, if there is one, to be 3. This is expressed by letting A be the predicate $h_a \preceq <(a,3)>$, where $\preceq$ denotes prefix relation. Then we know that the value passed to process P_2 is 9, and the value sent back to P_1 is 729. The program satisfies the specification

$$\begin{array}{ll} p:\ h_{abc} = <>, & q:\ x = 3 \wedge y = 729 \wedge z = 9, \\ A: h_a \preceq <(a,3)>, & C: h_b \preceq <(b,9)> \wedge h_c \preceq <(c,729)> \end{array}$$

and this can be proved by using the parallel composition rule. Indeed, P_1 satisfies

$$\begin{array}{ll} p:\ \ h_{abc} = <>, & q_1:\ x = 3 \wedge y = 729, \\ A_1: h_a \preceq <(a,3)> \wedge h_c \preceq <(c,729)>, & C_1: h_b \preceq <(b,9)>, \end{array}$$

and P_2 satisfies

$$\begin{array}{ll} p:\ \ h_{abc} = <>, & q_2:\ z = 9, \\ A_2: h_b \preceq <(b,9)>, & C_2: h_c \preceq <(c,729)>\ . \end{array}$$

Conclusion. Instead of surprising, the difference between the two parallel composition rules is probably what one would have expected. In short, this is due to different interaction mechanisms and different specified properties. However, this difference has for a long time cast doubts on whether it is possible to unify the two styles of concurrency, and in particular to have one general rule for both cases.

4 A General Rule

For a general rule to work, the assumption–commitment specifications cannot be restricted to the specific properties illustrated in the previous section. Now, specifications are built using temporal predicates directly (temporal merely in the sense that they are interpreted over behaviors). The notation $\sigma \models \varphi$ indicates that behavior σ satisfies predicate φ. A temporal predicate φ is a *safety* predicate iff for any behavior σ:

$$\sigma \models \varphi \qquad \text{iff} \qquad \text{for any finite } k \text{ such that } 0 \leq k \leq |\sigma| : \sigma|_k \models \varphi$$

A state predicate p is lifted to a temporal predicate by defining $\sigma \models p$ as $\sigma_0 \models p$. The boolean connectors $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ are also extended to temporal operands in the obvious way. For instance,

$$\sigma \models \varphi_1 \rightarrow \varphi_2 \qquad \text{iff} \qquad \text{if } \sigma \models \varphi_1 \text{ then } \sigma \models \varphi_2$$

An assumption-commitment specification is of the form $\mu : (E, < M^S, M^R >)$ where μ is a set of agents and E, M^S, M^R are temporal predicates. The assumption about the environment is described by E, which is restricted to be a safety predicate. The commitment is divided into a safety predicate M^S and a remaining predicate M^R. Liveness should be described by M^R, but for the soundness of the composition rule it is not necessary to insist on M^R to be a pure liveness predicate. The intended meaning of a specification is that a valid behavior satisfies the commitment provided that it satisfies the assumption. However, to eliminate the danger of circular reasoning when composing specifications without imposing extra conditions about specification predicates (as in [1] e.g.), we employ a slightly more complicated interpretation, based not only on logical implication $\rightarrow$ but also on a temporal operator $\hookrightarrow$. This so-called 'spiral' interpretation is used widely in message passing concurrency [12, 14, 20]; in shared variable concurrency, this interpretation is equivalent (as proved in [7]) to the one based on logical implication only [18, 19]. Let :

$$\sigma \models \varphi_1 \hookrightarrow \varphi_2 \qquad \text{iff} \qquad \begin{array}{l} \text{for any finite } k \text{ such that } 1 \leq k \leq |\sigma| : \\ \text{if } \sigma|_{k-1} \models \varphi_1 \text{ then } \sigma|_k \models \varphi_2 \end{array}$$

We then define :

$$\begin{array}{l} \models P \ \underline{sat} \ \ \mu : (E, \ < M^S, M^R >) \\ \text{iff} \\ \text{for all } \sigma \text{ in } [\![\mu : P]\!] \ : \ \sigma \models (E \hookrightarrow M^S) \ \wedge \ (E \rightarrow M^R) \end{array}$$

Therefore, for any behavior of P,

I′) if any prefix of σ satisfies E, then M^S holds after the transition extending that prefix, and

II′) if σ satisfies E, then σ also satisfies M^R.

A specification $\mu\!:\!(E, < M^S, M^R >)$ should also be μ–abstract: formally, for two behaviors σ and σ' such that $\sigma \simeq_\mu \sigma'$, this implies the following:

$$\sigma \models (E \hookrightarrow M^S) \ \wedge \ (E \rightarrow M^R) \quad \text{iff} \quad \sigma' \models (E \hookrightarrow M^S) \ \wedge \ (E \rightarrow M^R)$$

A syntactic composition rule for assumption–commitment specifications is:

$$\frac{\begin{array}{ll} \mathit{init}(E \rightarrow E_1 \wedge E_2) & \text{(PG1)} \\ E \wedge M_1^S \wedge M_2^S \rightarrow E_1 \wedge E_2 & \text{(PG2)} \\ M_1^S \wedge M_2^S \rightarrow M^S & \text{(PG3)} \\ E \wedge M_1^R \wedge M_2^R \rightarrow M^R & \text{(PG4)} \\ P_i \ \ \underline{sat} \ \ \mu_i : (E_i, \ \ < M_i^S, M_i^R >) & \text{(PG5)} \end{array}}{P_1 \parallel P_2 \ \ \underline{sat} \ \ \mu : (E, \ \ < M^S, M^R >)}$$

where $\mu = \mu_1 \cup \mu_2$, $\mu_1 \cap \mu_2 = \emptyset$, and operator *init* is defined as

$$\sigma \models \mathit{init}\, \varphi \qquad \text{iff} \qquad \sigma_0 \models \varphi$$

Our formulation is based mainly on [1] and the subsequent investigation [7] of it. Recently we were informed about an early paper by Barringer and Kuiper [3] that we did not notice when the current research was conducted. They already suggested to divide commitments into safety and liveness parts, and used a similar interpretation for the correctness formula. Moreover, our rule above is also similar to theirs. However, they only studied shared variable concurrency in that paper, and subsequently restricted the assumptions to state transitions as those used in the shared variable rule. The latest work by Abadi and Lamport [2] also uses 'spiral' interpretation. Instead of agents, the index variables in the TLA formulas seem to have played a similar role.

Informally, the soundness of our rule can be understood as follows. The first thing to observe is that both E_1 and E_2 hold as long as (i.e., will not become false before) E does. If E holds initially, then by PG1, E_1 and E_2 also hold. Suppose a transition is made (either by P_1, P_2 or the overall environment) leading to a new state. Then from PG5 it follows that M_1^S and M_2^S hold in the behavior up to the second state, and therefore, if E holds also up to the second state then from PG2 it follows that E_1 and E_2 hold too. We can repeat the argument over the next transition until all finite prefixes are considered. Because E, E_1 and E_2 are safety predicates, the complete (possibly infinite) behavior also satisfies E_1 and E_2 if it satisfies E. Now proving the soundness of the rule becomes straightforward: if any prefix of a behavior satisfies E, then we have just indicated that it satisfies E_1 and E_2, and consequently it follows from PG5 and PG3 that M^S holds after one more transition; if any complete behavior satisfies E, then we know that it satisfies E_1 and E_2, and therefore it follows from PG5 and PG4 that M^R also holds. A formal proof is given below.

Proof ((Soundness of the general rule)).
We want to prove $\models P_1 \parallel P_2\ \underline{sat}\ \ \mu : (E,\ \ < M^S, M^R >)$ under the assumption that PG1–PG5 are valid. We first establish a lemma (which only relies on PG1 and PG2):

i) $\models E \wedge (E_1 \hookrightarrow M_1^S) \wedge (E_2 \hookrightarrow M_2^S) \rightarrow E_1 \wedge E_2$. Let us assume $\sigma \models E$ and $\sigma \models (E_1 \hookrightarrow M_1^S) \wedge (E_2 \hookrightarrow M_2^S)$; since E_1 and E_2 are safety predicates, we only have to prove $\sigma|_k \models E_1 \wedge E_2$ for any finite k such that $0 \leq k \leq |\sigma|$. This proceeds by induction on k.

(a) base: $k = 0$

(1)	$\sigma \models E$	assumption
(2)	$\sigma|_0 \models E$	E is safety, (1)
(3)	$\sigma|_0 \models E_1 \wedge E_2$	(2), PG1

(b) induction: assume $\sigma|_{k-1} \models E_1 \wedge E_2$.

(1)	$\sigma \models E$	assumption
(2)	$\sigma|_k \models E$	E is safety, (1)
(3)	$\sigma \models (E_1 \hookrightarrow M_1^S) \wedge (E_2 \hookrightarrow M_2^S)$	assumption
(4)	$\sigma|_{k-1} \models E_1 \wedge E_2$	ind. hyp.
(5)	$\sigma|_k \models M_1^S \wedge M_2^S$	(3), (4)
(6)	$\sigma|_k \models E_1 \wedge E_2$	(2), (4), PG2

We have $\llbracket \mu : P_1 \parallel P_2 \rrbracket = (\llbracket \mu_1 : P_1 \rrbracket \cap \llbracket \mu_2 : P_2 \rrbracket)^{\simeq_\mu}$. Because the specification $\mu : (E,\ \ < M^S, M^R >)$ is μ–abstract, we only have to check that for any behavior $\sigma \in \llbracket \mu_1 : P_1 \rrbracket \cap \llbracket \mu_2 : P_2 \rrbracket$, the following ii) and iii) hold

ii) $\sigma \models E \hookrightarrow M^S$. For any finite k such that $1 \leq k \leq |\sigma|$, we assume $\sigma|_{k-1} \models E$ and prove $\sigma|_k \models M^S$.

(1)	$\sigma|_{k-1} \models E$	assumption
(2)	$\sigma \in \llbracket \mu_1 : P_1 \rrbracket \cap \llbracket \mu_2 : P_2 \rrbracket$	assumption
(3)	$\sigma \models (E_1 \hookrightarrow M_1^S) \wedge (E_2 \hookrightarrow M_2^S)$	(2), PG5
(4)	$\sigma|_{k-1} \models (E_1 \hookrightarrow M_1^S) \wedge (E_2 \hookrightarrow M_2^S)$	(3), $E_i \hookrightarrow M_i^S$ are safety
(5)	$\sigma|_{k-1} \models E_1 \wedge E_2$	(1), (4), i)
(6)	$\sigma|_k \models M_1^S \wedge M_2^S$	(3), (5)
(7)	$\sigma|_k \models M^S$	(6), PG3

iii) $\sigma \models E \rightarrow M^R$. We assume $\sigma \models E$ and prove $\sigma \models M^R$.

(1)	$\sigma \models E$	assumption
(2)	$\sigma \in \llbracket \mu_1 : P_1 \rrbracket \cap \llbracket \mu_2 : P_2 \rrbracket$	assumption
(3)	$\sigma \models (E_1 \hookrightarrow M_1^S) \wedge (E_2 \hookrightarrow M_2^S)$	(2), PG5
(4)	$\sigma \models E_1 \wedge E_2$	(1), (3), i)
(5)	$\sigma \models (E_1 \rightarrow M_1^R) \wedge (E_2 \rightarrow M_2^R)$	(2), PG5
(6)	$\sigma \models M_1^R \wedge M_2^R$	(4), (5)
(7)	$\sigma \models M^R$	(1), (6), PG4

□

5 Unification Proof

Having established the general rule, the remaining part of this paper is to show that the two rules for shared variable and `OCCAM`-like programs are special cases of it. To match the schema of the general rule, we have to map the tuple $(p,\ A,\ C,\ q)$ to a specification of the form $\mu : (E, < M^S, M^R >)$, so that points I′) and II′) of the general rule in Sect. 4 capture the points I) and II) of the specific rules in Sect. 3. It clearly appears from the comparison of these points that:

– E is the conjunction of p (lifted to a temporal predicate: $\sigma \models p$ iff $\sigma_0 \models p$) with another temporal predicate that captures the meaning of A.
– M^S is a temporal predicate that captures the meaning of C.
– M^R is the conjunction of the temporal predicates fin_μ and $post\ q$, where

$\sigma \models fin_\mu$	iff	the number of μ transitions in σ is finite
$\sigma \models post\ q$	iff	q holds for any terminated (ticked) state of σ

We shall be more precise about the temporal predicates that capture the meaning of A and C when we come to the two particular proof rules. We first list a number of useful algebraic properties of the operators *init*, *post* and *fin*. These operators can be viewed as functions from predicates to predicates. Like usual functional operators, we assume they bind more closely than boolean connectors.

$$\models\ init\, t_1 \ \rightarrow\ init\, t_2 \quad \text{if} \quad \models\ t_1 \rightarrow t_2 \qquad \text{(I1)}$$
$$\models\ init\ (t_1 \wedge t_2)\ \leftrightarrow\ init\, t_1\ \wedge\ init\, t_2 \qquad \text{(I2)}$$
$$\models\ init\ (t_1 \rightarrow t_2)\ \leftrightarrow\ (init\, t_1\ \rightarrow\ init\, t_2) \qquad \text{(I3)}$$
$$\models\ fin_{\mu_1}\ \wedge\ fin_{\mu_2}\ \leftrightarrow\ fin_{\mu_1 \cup \mu_2} \qquad \text{(F1)}$$
$$\models\ post\, q_1\ \wedge\ post\, q_2\ \leftrightarrow\ post\ (q_1 \wedge q_2) \qquad \text{(Q1)}$$

5.1 Shared Variable Concurrency

In order to transform binary state predicates A and C of Sect. 3.1 into temporal predicates, we introduce a new temporal operator *links*. Let μ be a set of agents and r be a binary state predicate:

$\sigma \models links_\mu\ r$
iff
for any transition $\sigma_i \xrightarrow{\delta} \sigma_{i+1}$ such that $\delta \in \mu,\ (\sigma_i, \sigma_{i+1}) \models r$.

Operator *links* enjoys the following algebraic properties:

$$\models\ links_\mu\, r_1\ \rightarrow\ links_\mu\, r_2, \quad \text{if} \quad \models\ r_1 \rightarrow r_2 \qquad \text{(L1)}$$
$$\models\ links_{\mu_1}\, r\ \wedge\ links_{\mu_2}\, r\ \leftrightarrow\ links_{\mu_1 \cup \mu_2}\, r \qquad \text{(L2)}$$
$$\models\ init\ (links_\mu\ r) \qquad \text{(IL)}$$

Since environment and component transitions in $[\![\mu : P]\!]$ are labeled with agents in $\overline{\mu}$ and μ respectively, the second conjunct of E is $links_{\overline{\mu}}\ A$ whereas M^S is the predicate $links_\mu C$. In summary, the specification tuples (p, A_1, C_1, q_1),

(p, A_2, C_2, q_2), and $(p, A, C, q_1 \wedge q_2)$ of the shared variable rule are mapped respectively to:

$$\mu_1 : (E_1, < M_1^S, M_1^R >) \qquad \text{where} \qquad \begin{cases} E_1 \overset{\text{def}}{=} p \wedge links_{\overline{\mu_1}}\ A_1 \\ M_1^S \overset{\text{def}}{=} links_{\mu_1}\ C_1 \\ M_1^R \overset{\text{def}}{=} post\ q_1 \wedge fin_{\mu_1} \end{cases}$$

$$\mu_2 : (E_2, < M_2^S, M_2^R >) \qquad \text{where} \qquad \begin{cases} E_2 \overset{\text{def}}{=} p \wedge links_{\overline{\mu_2}}\ A_2 \\ M_2^S \overset{\text{def}}{=} links_{\mu_2}\ C_2 \\ M_2^R \overset{\text{def}}{=} post\ q_2 \wedge fin_{\mu_2} \end{cases}$$

$$\mu : (E, < M^S, M^R >) \qquad \text{where} \qquad \begin{cases} E \overset{\text{def}}{=} p \wedge links_{\overline{\mu}}\ A \\ M^S \overset{\text{def}}{=} links_{\mu}\ C \\ M^R \overset{\text{def}}{=} post\ (q_1 \wedge q_2) \wedge fin_{\mu} \end{cases}$$

We then derive the shared variable rule from the general one by showing that the premises in the general rule (listed on the left side of the table below) are implied by the premises in the shared variable rule (listed on the right side and labeled by PS1–PS3).

$\models init(E \rightarrow E_1 \wedge E_2)$	$\models A \vee C_1 \rightarrow A_2$	(PS1)
$\models E \wedge M_1^S \wedge M_2^S \rightarrow E_1 \wedge E_2$	$\models A \vee C_2 \rightarrow A_1$	(PS2)
$\models M_1^S \wedge M_2^S \rightarrow M^S$	$\models C_1 \vee C_2 \rightarrow C$	(PS3)
$\models E \wedge M_1^R \wedge M_2^R \rightarrow M^R$		

In particular, the use of *disjunction* in PS1 (and similarly for PS2, PS3) is due to the following fact:

$$\star \text{ if } \models A \vee C_1 \rightarrow A_2 \text{ then } \models links_{\overline{\mu}}\ A \wedge links_{\mu_1}\ C_1 \rightarrow links_{\overline{\mu_2}}\ A_2$$

This follows directly from L1, L2, and the observation $\overline{\mu_2} = \overline{\mu} \cup \mu_1$ (because $\mu = \mu_1 \cup \mu_2$ and $\mu_1 \cap \mu_2 = \emptyset$).

Premise 1: $\models init(E \rightarrow E_1 \wedge E_2)$

(1)	$\models init\ E \leftrightarrow init\ p$	definition, I2, IL
(2)	$\models init\ E_1 \leftrightarrow init\ p$	definition, I2, IL
(3)	$\models init\ E_2 \leftrightarrow init\ p$	definition, I2, IL
(4)	$\models init\ E \rightarrow init\ E_1 \wedge init\ E_2$	(1)–(3)
(5)	$\models init(E \rightarrow E_1 \wedge E_2)$	(4), I2, I3

Premise 2: $\models E \wedge M_1^S \wedge M_2^S \rightarrow E_1 \wedge E_2$.
We prove $\models E \wedge M_1^S \rightarrow E_2$. The proof of $\models E \wedge M_2^S \rightarrow E_1$ is similar.

(1)	$\models links_{\overline{\mu}}\ A \rightarrow links_{\overline{\mu}}\ A_2$	PS1, L1
(2)	$\models links_{\mu_1}\ C_1 \rightarrow links_{\mu_1}\ A_2$	PS1, L1
(3)	$\models links_{\overline{\mu}}\ A_2 \wedge links_{\mu_1}\ A_2 \rightarrow links_{\overline{\mu_2}}\ A_2$	$\overline{\mu} \cup \mu_1 = \overline{\mu_2}$, L2
(4)	$\models links_{\overline{\mu}}\ A \wedge links_{\mu_1}\ C_1 \rightarrow links_{\overline{\mu_2}}\ A_2$	(1)–(3)
(5)	$\models E \wedge M_1^S \rightarrow E_2$	(4), definitions

Premise 3: $\models M_1^S \wedge M_2^S \rightarrow M^S$.

(1)	$\models$ *links*$_{\mu_1}$ $C_1 \rightarrow$ *links*$_{\mu_1}$ C	PS3, L1
(2)	$\models$ *links*$_{\mu_2}$ $C_2 \rightarrow$ *links*$_{\mu_2}$ C	PS3, L1
(3)	$\models$ *links*$_{\mu_1}$ $C \wedge$ *links*$_{\mu_2}$ $C \rightarrow$ *links*$_{\mu}$ C	$\mu_1 \cup \mu_2 = \mu$, L2
(4)	$\models$ *links*$_{\mu_1}$ $C_1 \wedge$ *links*$_{\mu_2}$ $C_2 \rightarrow$ *links*$_{\mu}$ C	(1)–(3)
(5)	$\models M_1^S \wedge M_2^S \rightarrow M^S$	(4), definitions

Premise 4: $\models E \wedge M_1^R \wedge M_2^R \rightarrow M^R$. We prove $\models M_1^R \wedge M_2^R \rightarrow M^R$.

(1)	$\models$ *post* $q_1 \wedge$ *post* $q_2 \rightarrow$ *post* $(q_1 \wedge q_2)$	Q1
(2)	$\models fin_{\mu_1} \wedge fin_{\mu_2} \rightarrow fin_{\mu}$	$\mu_1 \cup \mu_2 = \mu$, F1
(3)	$\models M_1^R \wedge M_2^R \rightarrow M^R$	(1), (2), definitions

5.2 Message Passing Concurrency

In order to transform the trace predicates A and C of Sect. 3.2 into temporal predicates, we apply the usual 'always' operator $\Box$ of temporal logic. Notice that, since states record communication traces, trace predicates can be viewed as state predicates. For any trace predicate r, let:

$$\sigma \models \Box\, r \quad \text{iff} \quad \sigma_i \models r, \;\; \text{for any finite } i \text{ such that } 0 \leq i \leq |\sigma|$$

The 'always' operator enjoys the following properties:

$\models \Box\, r_1 \rightarrow \Box\, r_2, \quad \text{if} \quad \models r_1 \rightarrow r_2$	($\Box$1)
$\models \Box\,(r_1 \wedge r_2) \leftrightarrow \Box\, r_1 \wedge \Box\, r_2$	($\Box$2)
$\models init\; \Box\, r \leftrightarrow init\; r$	(IA)

Thus, the second conjunct of E is $\Box\, A$, and M^S is $\Box\, C$. The specification tuples (p, A_1, C_1, q_1), (p, A_2, C_2, q_2), and $(p, A, C, q_1 \wedge q_2)$ of the `OCCAM` rule are mapped respectively to:

$$\mu_1 : (E_1, < M_1^S, M_1^R >) \qquad \text{where} \qquad \begin{cases} E_1 \overset{\text{def}}{=} p \wedge \Box A_1 \\ M_1^S \overset{\text{def}}{=} \Box C_1 \\ M_1^R \overset{\text{def}}{=} post\; q_1 \wedge fin_{\mu_1} \end{cases}$$

$$\mu_2 : (E_2, < M_2^S, M_2^R >) \qquad \text{where} \qquad \begin{cases} E_2 \overset{\text{def}}{=} p \wedge \Box A_2 \\ M_2^S \overset{\text{def}}{=} \Box C_2 \\ M_2^R \overset{\text{def}}{=} post\; q_2 \wedge fin_{\mu_2} \end{cases}$$

$$\mu : (E, < M^S, M^R >) \qquad \text{where} \qquad \begin{cases} E \overset{\text{def}}{=} p \wedge \Box A \\ M^S \overset{\text{def}}{=} \Box C \\ M^R \overset{\text{def}}{=} post\; (q_1 \wedge q_2) \wedge fin_{\mu} \end{cases}$$

In the same vein as before, we derive the `OCCAM` rule by showing that its premises imply those of the general rule:

$\models init(E \rightarrow E_1 \wedge E_2)$	$\models A \wedge C_1 \rightarrow A_2$	(PM1)
$\models E \wedge M_1^S \wedge M_2^S \rightarrow E_1 \wedge E_2$	$\models A \wedge C_2 \rightarrow A_1$	(PM2)
$\models M_1^S \wedge M_2^S \rightarrow M^S$	$\models C_1 \wedge C_2 \rightarrow C$	(PM3)
$\models E \wedge M_1^R \wedge M_2^R \rightarrow M^R$	$\models p \wedge A \rightarrow A_1 \wedge A_2$	(PM4)

The use of conjunction in PM1 (and similarly for PM2, PM3) is due to the fact:

$\star$ if $\models A \wedge C_1 \rightarrow A_2$ then $\models \Box A \wedge \Box C_1 \rightarrow \Box A_2$

which follows directly from $\Box 1$ and $\Box 2$.

Premise 1: $\models init(E \rightarrow E_1 \wedge E_2)$

(1)	$\models init\ E \leftrightarrow init\ (p \wedge A)$	definition, I2, IA
(2)	$\models init\ E_1 \leftrightarrow init\ (p \wedge A_1)$	definition, I2, IA
(3)	$\models init\ E_2 \leftrightarrow init\ (p \wedge A_2)$	definition, I2, IA
(4)	$\models init\ E \rightarrow init\ E_1 \wedge init\ E_2$	(1)–(3), I1, PM4
(5)	$\models init(E \rightarrow E_1 \wedge E_2)$	(4), I2, I3

Premise 2: $\models E \wedge M_1^S \wedge M_2^S \rightarrow E_1 \wedge E_2$. We prove $\models E \wedge M_1^S \rightarrow E_2$. The proof of $\models E \wedge M_2^S \rightarrow E_1$ is similar.

(1)	$\models \Box A \wedge \Box C_1 \rightarrow \Box A_2$	$\Box 1$, $\Box 2$, PM1
(2)	$\models E \wedge M_1^S \rightarrow E_2$	(1), definitions

Premise 3: $\models M_1^S \wedge M_2^S \rightarrow M^S$.

(1)	$\models \Box C_1 \wedge \Box C_2 \rightarrow \Box C$	$\Box 1$, $\Box 2$, PM3
(2)	$\models M_1^S \wedge M_2^S \rightarrow M^S$	(1), definitions

Premise 4: $\models E \wedge M_1^R \wedge M_2^R \rightarrow M^R$. Same as for the shared variable case.

6 Discussion

Cliff Jones is probably the first person who noticed the similarity (and of course the obvious difference) between the two proof rules for shared variable and `OCCAM`-like concurrent programs. Furthermore, he remarked in [9], "the extension of a modal logic to cover binary relations may yield some interesting insights. In particular, more general forms of rely- and guarantee- conditions should be definable, and perhaps some unification of proof methods found".

This paper has made one step in exactly this direction. Although the general rule does not contain any modal operators directly, we base it on temporal logic, because the formulae are interpreted over, in principle, possibly infinite behaviors. To facilitate the derivations of the two rules for shared variable and `OCCAM`-like concurrent programs, we (inspired by Jones' recent work, e.g., [10]) have chosen application–oriented temporal operators which are defined semantically. It is possible to use a standard temporal logic (e.g., [11]) instead, but then the derivations in this paper would become less straightforward.

Being a unified rule, the general rule can be used for both shared variable and message passing concurrency, and as a matter of fact, for any other model which satisfies the general properties we have discussed. However, most of the applications that we are aware of in concurrency are based on one single communication model. In such cases, the direct use of a general rule is not recommended: it is much more effective to use the specialized rules for specific verification tasks. The general rule, on the other hand, may be useful in models in which shared variable concurrency and message passing concurrency are jointly combined. For example, the distributed mutual exclusion algorithm reported in [4] uses both shared variables and message passing, and therefore it could be a good test for our general rule. It must be noted, however, that understanding a complicated algorithm is always a difficult task, and a formal verification is certain to be a challenge, to which a good method can only provide a (relatively) effective tool, but never a ready solution.

Acknowledgement We are grateful to Willem-Paul de Roever for repeatedly drawing our attention to the problem, much encouragement, and extensive comments. We thank Cliff Jones for a careful reading and comments which led to a major restructuring of the presentation. A discussion with Job Zwiers, and comments from the anonymous referees were also very helpful.

References

1. M. Abadi and L. Lamport. Composing specifications. ACM Trans. on Program. Lang. Syst., 15:73–132, 1993.
2. M. Abadi and L. Lamport. Conjoining specifications. Digital Equipment Corporation Systems Research Center, Research Report 118, 1993.
3. H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In S.D. Brookes, A.W. Roscoe and G. Winskel eds., Proc. of Seminar on Concurrency 1984, LNCS 197, Springer-Verlag, 1985.
4. M. Ben-Ari. Principles of Concurrent and Distributed Programming. Chapter 11. Prentice Hall, 1990.
5. P. Collette. Application of the composition principle to Unity–like specifications. In M.-C. Gaudel and J.-P. Jouannaud eds., Proc. of TAPSOFT 93, LNCS 668, Springer-Verlag, 1993.
6. P. Collette. An explanatory presentation of composition rules for assumption-commitment specifications. Information Processing Letters, 50:31–35, 1994.
7. P. Collette and A. Cau. Parallel composition of Assumption–Commitment specifications: a unifying approach for shared variable and distributed message passing concurrency. Technical Report 94-03, Université Catholique de Louvain, 1994.
8. C.B. Jones. Development methods for computer programs including a notion of interference. DPhil. Thesis, Oxford University Computing Laboratory, 1981.

9. C.B. Jones. Tentative steps towards a development method for interfering programs. ACM Trans. Program. Lang. Syst., 5(4):596–619, 1983.
10. C.B. Jones. Interference resumed. In P. Baile ed., Australian Software Engineering Research, 1991.
11. Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems, specification, Vol. I. Springer-Verlag, 1991.
12. J. Misra and M. Chandy. Proofs of Networks of Processes. IEEE SE, 7(4):417-426, 1981.
13. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. Acta Inform., 6:319–340, Springer-Verlag, 1976.
14. P.K. Pandya and M. Joseph. P-A logic - a compositional proof system for distributed programs. Distributed Computing, 5:37–54, 1991
15. W.-P. de Roever. The quest for compositionality. Proc. of IFIP Working Conf., The Role of Abstract Models in Computer Science, North-Holland, 1985.
16. W.-P. de Roever, J. Hooman, F. de Boer, Y. Lakhneche, Q. Xu and P. Pandya. State-Based Proof Theory of Concurrency: from noncompositional to compositional methods. Book manuscript, 350 pages, Christian-Albrechts-Universität zu Kiel, Germany, 1994.
17. C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. Theoretical Computer Science, 58:347–359, 1988.
18. K. Stølen. A method for the development of totally correct shared-state parallel programs. In J.C.M. Baeten and J.F. Groote eds., Proc. of CONCUR 91, LNCS 527, Springer-Verlag, 1991.
19. Q. Xu and J. He. A theory of state-based parallel programming: Part 1. In J. Morris and R. Shaw eds., Proc. of BCS FACS 4th Refinement Workshop, Cambridge, Springer-Verlag, 1991.
20. J. Zwiers, A. de Bruin and W.-P. de Roever. A proof system for partial correctness of Dynamic Networks of Processes. Proc. of the Conference on Logics of Programs 1983, LNCS 164, Springer-Verlag, 1984.