A foundation for modular reasoning about safety and progress properties of state-based concurrent programs¹

Pierre Collette*, Edgar Knapp

Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK

Abstract

This paper introduces open systems with non-trivial environment actions and proposes a cooperation condition for composing them. The analysis of the condition, expressed in terms of predicate transformers, leads to a stepwise and explanatory construction of parallel composition rules. The completeness of a proof system for the compositional verification of UNITY programs is then established.

1. Introduction

Modular approaches to software verification and software development require composition rules for verifying the correctness of a composite module from the correctness of its components (bottom-up design) and conversely for validating the decomposition of a module specification into specifications of components that are easier to implement (top-down design). Examples of the so-called compositional proof systems are given in [4, 19] (temporal logic specifications), [20, 26] (trace specifications in message-based concurrency), and [24] (extended pre/post specifications for state-based concurrency). Following the same lines, this paper presents composition rules that support the modular verification and development of state-based concurrent programs in a UNITY-like style. Designed by Chandy and Misra [5], the UNITY formalism (programs and logic) has proved useful in the formal specification, development, and verification of concurrent programs; examples are [16, 23].

Program components are viewed as open systems where component steps are interleaved with environment steps (steps from other components). The properties of a component then depend on the assumptions that can be made on these environment steps. More precisely, a component is said to satisfy a property \((r, \Phi)\) if it satisfies \(\Phi\)

* Corresponding author.

¹ Work supported by the UK Engineering and Physical Sciences Research Council.

0304-3975/97/$17.00 © 1997 Published by Elsevier Science B.V. All rights reserved

PII S0304-3975(96)00327-1
when interacting with an environment whose steps satisfy the rely condition \( r \). Modular reasoning is thus achieved by applying the assumption-commitment paradigm of e.g. [12, 18, 21].

One important result in this paper is the compositional completeness of the proposed proof system for UNITY. As shown in [4], compositional completeness for the verification of temporal properties of state-based concurrent programs can be achieved in other ways but the inherent simplicity of the UNITY logic has then to be given up for a richer but barely workable temporal logic. In contrast, compositional completeness is achieved whilst preserving the elegance and workability of the formalism; the only extra cost is the extension of program properties with (a precondition and) a rely condition. This simple modification to the logic also solves a number of deficiencies of the previous variants of the UNITY logic; this is further discussed in Section 9. Another novelty of this paper w.r.t. other work on assumption-commitment specifications is its devotion to a presentation based on predicate transformers and extreme solutions of equations in predicates. This of course is only a matter of style but the definitions seem to be more workable, the motivation for the theorems clearer, the formulas simpler, and the proofs more calculational. In brief, the often tedious task of reasoning with program computations is avoided.

This paper is a fully revised and expanded version of [9]. To factor out the fundamental results (especially on program composition), language-specific features have been abstracted away. Instead of being expressions in a first-order language, predicates are viewed as functions from states to booleans; program statements are then replaced by actions (universally conjunctive predicate transformers) and so is the rely condition \( r \) in specifications. Since predicates and actions can be ordered in lattices, this abstraction yields an elegant algebraic style of reasoning. The second major difference with [9] is the formal definition of open systems as a generalisation of UNITY programs. Thanks to the introduction of environment actions, this generalisation allows assumption-commitment properties to be given a simple workable axiomatic semantics: a system \( O \) has the property \((r, \Phi)\) if the system \( O \) augmented with the environment action \( r \) has the property \( \Phi \). This new way of understanding assumption-commitment properties is especially enlightening when discussing the soundness and completeness of the parallel composition rules.

The central part of this paper consists of Sections 5 and 6. First, definitions and notational conventions for predicates and predicate transformers are given in Section 2. Open systems and their properties are then defined in Sections 3 and 4. Sections 7–9 are devoted to more advanced issues.

2. Predicates and actions

This section follows [3] to present a formalisation of predicates and actions in a lattice theory setting. Actions are defined as the subclass of predicate transformers that give weakest preconditions for relations, and therefore for the nondeterministic
In the sequel, infima and suprema of sets of actions will be used to combine component and environment actions, respectively.

By convention, \(\cdot\) denotes functional application; this operator has the highest binding power and associates to the left; for instance, \(f.a.b\) is parsed as \((f(a))(b)\). Grouped by decreasing order of binding power, the other operators introduced in this section are: \((\neg, (\cdot)^{\circ}, \downarrow, \uparrow), (\lor, \land, \lor, \land, \lor), (\Rightarrow, \subseteq, \subseteq), =, \text{ and } \equiv\). The operators \(\neg, \lor, \land, \Rightarrow\), and \(\equiv\) are strictly reserved for boolean expressions. Expressions with quantifiers are surrounded by brackets with colons to separate the dummies, the range, and the quantified expression.

A state is a function from \(\text{VAR}\) to values where \(\text{VAR}\) denotes the set of all programming variables. The set of all states is denoted by \(\Sigma\); by convention, \(\sigma\) and \(\tau\) range over \(\Sigma\). A predicate is a function from \(\Sigma\) to the set \{true, false\} of booleans. Predicates are equivalently viewed as subsets of \(\Sigma\); in particular \{\(\sigma\)\} is the predicate that holds in \(\sigma\) only. A relation is a function from \(\Sigma \times \Sigma\) to booleans. By convention, \(P, Q, X\) (the unknown in equations) range over predicates, and \(R\) ranges over relations; for convenience, \(\sigma R \tau\) stands for \(R.(\sigma, \tau)\). Predicates and relations over \(V\) (\(\subseteq \text{VAR}\)) do not distinguish between states related by the relation \(=_{V}\); the projection of \(R\) onto \(V\) is denoted by \(R/V\):

\[
\sigma =_{V} \tau \equiv \langle \forall v : v \in V : \sigma.v = \tau.v \rangle
\]

\(P\) is over \(V\) \(\equiv\) \(\langle \forall \sigma, \tau : \sigma =_{V} \tau : P.\sigma \equiv P.\tau \rangle\)

\(R\) is over \(V\) \(\equiv\) \(\langle \forall \sigma, \sigma', \tau, \tau' : \sigma =_{V} \sigma' \land \tau =_{V} \tau' : \sigma R \tau \equiv \sigma' R \tau' \rangle\)

\(\sigma(R/V)\tau \equiv \langle \exists \sigma', \tau' : \sigma =_{V} \sigma' \land \tau =_{V} \tau' : \sigma' R \tau' \rangle\)

The order \(\subseteq\) on predicates is the pointwise extension of the order \(\Rightarrow\) on booleans:

\(P \subseteq Q \equiv \langle \forall \sigma : P.\sigma \Rightarrow Q.\sigma \rangle\)

With this order, the set of predicates is a complete boolean lattice. The top element \(\text{True}\) is defined by \(\text{True}.\sigma \equiv \text{true}\); the bottom element \(\text{False}\) is defined by \(\text{False}.\sigma \equiv \text{false}\); the complement \(P^{c}\) of \(P\) is defined by \(P^{c}.\sigma \equiv \neg P.\sigma\). Suprema (\(\cup\)) and infima (\(\cap\)) are defined (pointwise and for any set) by

\[
(P \cup Q).\sigma \equiv P.\sigma \lor Q.\sigma \quad (\bigcup\{i : i \in I : P_{i}\}).\sigma \equiv (\exists i : i \in I : P_{i}.\sigma)
\]

\[
(P \cap Q).\sigma \equiv P.\sigma \land Q.\sigma \quad (\bigcap\{i : i \in I : P_{i}\}).\sigma \equiv (\forall i : i \in I : P_{i}.\sigma)
\]

The complete boolean lattice of relations is defined in a similar way.

A predicate transformer is a function from predicates to predicates. The order \(\subseteq\) on predicate transformers is the pointwise extension of the order \(\subseteq\) on predicates:

\(f \subseteq g \equiv \langle \forall P : f.P \subseteq g.P \rangle\)
A predicate transformer \( f \) is **monotonic** if \( f.P \subseteq f.Q \) whenever \( P \subseteq Q \); \( f \) is **universally conjunctive** (hence monotonic) if

\[
f.(\bigcap i: i \in I: P_i) = (\bigcap i: i \in I: f.P_i)
\]

In particular, for \( I \) empty, \( f.\text{True} = \text{True} \). In this paper, universally conjunctive predicate transformers are called **actions**; by convention, \( a, b \) range over actions. The set of actions ordered by \( \subseteq \) is a complete lattice which mirrors the lattice of relations. The function \( [.] \) maps actions to relations and its inverse \( [.] \) maps relations to actions; as the definition of \( [.] \) shows, actions give weakest (liberal) preconditions for relations

\[
[R].P.\sigma \equiv (\forall \tau: \sigma R \tau: P.\tau)
\]

\[\sigma[a] \tau \equiv \neg(a.\{\tau\}^c.\sigma)\]

Since \( a = [a] \) (and conversely \( R = [R] \)), an action \( a \) is completely defined by the relation \( [a] \). In particular, infima (\( \cap \)) and suprema (\( \cup \)) are defined (pointwise and for any set) by intersection and union of relations, respectively,

\[
[a \cup b] = [a] \cap [b] \quad [\bigcup i: i \in I: a_i] = (\bigcap i: i \in I: [a_i])
\]

\[
[a \cap b] = [a] \cup [b] \quad [\bigcap i: i \in I: a_i] = (\bigcup i: i \in I: [a_i])
\]

Equivalently, \( (a \cap b).P = a.P \cap b.P \). Note that \( (a \cup b).P \) might be strictly greater than \( a.P \cup b.P \) because the predicate transformer that maps \( P \) to \( a.P \cup b.P \) is not necessarily an action. Because \( a \subseteq b \) is equivalent to \( [b] \subseteq [a] \), \( a \subseteq b \) also reads '\( b \) is a refinement of \( a \)' or '\( b \) is more deterministic than \( a \)''. The action \( a \) is said to be over \( V \), reflexive, or transitive if \( [a] \) is. The projection of \( a \) onto \( V \) is defined by \( [a]/V = [a]/V \). Actions can be combined with predicates to give new actions; \( a \uparrow P \) is equivalently defined by

\[
\sigma[a \uparrow P] \tau \equiv P.\sigma \land \sigma[a] \tau \quad \sigma[a \uparrow P] \tau \equiv P.\sigma \Rightarrow \sigma[a] \tau
\]

The set of **reflexive** actions ordered by \( \subseteq \) is also a complete lattice, which mirrors the lattice of reflexive relations. Infimas and suprema for non-empty sets of reflexive actions are given by \( \cap \) and \( \cup \), respectively. The top element (infima of the empty set) is the most deterministic reflexive action \( \text{skip} \); the bottom element (suprema of the empty set) is the least deterministic reflexive action \( \text{random} \):

\[
\sigma[\text{skip}] \tau \equiv \sigma = \tau \quad \sigma[\text{random}] \tau \equiv \text{true}
\]

Finally, an action \( a \) is non-miraculous if \( a.\text{False} = \text{False} \); equivalently, \( a \) is non-miraculous if for every \( \sigma \), there exists \( \tau \) such that \( \sigma[a] \tau \). In particular, reflexive actions are non-miraculous. All terminating (possibly nondeterministic) statements of a conventional programming language can be interpreted as (input/output) relations. This explains why the weakest precondition semantics \( \text{wp} \).\( S \) of a terminating statement \( S \) is a non-miraculous and universally conjunctive predicate transformer [10].
3. Open systems

State-based concurrent programs are built of program components that interact via shared variables. In an interleaving model of concurrency, a computation appears to each component as a sequence of states where each step (state transition) is either performed by that component or by the environment of that component, i.e. by other components. This motivates the introduction of open systems as systems that include environment steps in their computations.

3.1. Definition

An open system O consists of a predicate I.O that specifies initial states, a set Co.O of component actions, a subset Fa.O of Co.O of fair component actions that must be executed infinitely often, and an action e.O meant to represent the interference of the environment in the execution of the component. All actions of O are non-miraculous; in operational terms (cf. Section 2), this means that all actions are always enabled. The reflexivity of e.O (i.e. of the relation e.O) is required because environment steps cannot be forced to change the shared state. Technically, this also ensures that the suprema (⊔) of two environment actions is non-miraculous.


The action c.O represents the (demonic) nondeterministic choice between component actions. The reflexive action e.O ⊔ c.O therefore represents the nondeterministic choice between all (environment or component) actions of O. The only reason to include a set Co.O of actions rather than just the single-component action c.O is to be able to discuss fairness and therefore progress properties. In contrast, including a set of environment actions would lead to unnecessary complications when composing systems.


3.2. Computations

Operational considerations play only a supporting role in this paper. Their presentation is therefore less rigorous but the formal axiomatic definitions in the rest of this paper can always be linked to their operational interpretation via the bijection between actions and relations described in Section 2.

A computation is an infinite sequence of states σ₀ ⊨₁ σ₁ ⊨₂ σ₂ ⊨₃ ... where environment and component steps are labelled (lᵣ) with e and c, respectively. In a computation of O, steps labelled with e are constrained by e.O whereas steps labelled with c are constrained by c.O. Moreover, σ₀ must be an initial state and each action in Fa.O must be executed infinitely often (this is usually called unconditional fairness but there
Program $F$

Shared $x : \text{int}$
Local $y : \text{int}$
Initially $y = 1$

Assign
$x, y := 2x - y + 1, y + 1$
$| x : 2x$

Fig. 1. Example program.

are no difference with weak/strong fairness in this case because actions are always enabled).

A computation of $O$ is thus such that

$I. O. \sigma_0 \land (\forall k : k > 0 : l_k = e \Rightarrow \sigma_k \cap [e. O] \sigma_k)$

$\land (\forall k : k > 0 : l_k = e \Rightarrow \sigma_k \cap [e. O] \sigma_k)$

$\land (\forall a : a \in FA. O : (\forall k : k > 0 : (\exists j : j \geq k : l_j = e \land \sigma_{j-1}[a](j)))}$

A state is reachable in $O$ if it occurs in a computation of $O$. Other compositional models of programs based on computations with environment steps can be found in e.g. [4, 24]; details specific to UNITY programs are given in [7, 8, 25].

3.3. UNITY programs

UNITY programs are now presented as a special case of open systems. An example program is given in Fig. 1. The important difference with Chandy and Misra's UNITY programs [5] is the introduction of local variables. Unlike the usual shared variables of [5], local variables cannot be accessed by concurrently executed programs. The second, less important, difference with [5] is the absence of initial conditions on the shared variables. Instead, much like in Hoare triples, these initial conditions will appear as preconditions in specifications.

Definition 3. A UNITY program $F$ consists of:
- Two disjoint finite sets $\text{shr}. F$ of shared variables and $\text{loc}. F$ of local variables. The set of all variables is $\text{var}. F = \text{shr}. F \cup \text{loc}. F$.
- A state predicate $\text{init}. F$ over $\text{loc}. F$ different from False.
- A finite set $\text{asg}. F$ of assignment statements over $\text{var}. F$.

The fact that the statements in $\text{asg}. F$ are assignment statements is not essential. In the sequel, it is only assumed that $S \in \text{asg}. F$ is a terminating statement over $\text{var}. F$ whose semantics is given by $\text{wp}. S$; $\text{wp}. S. P$ is the weakest precondition for $S$ to terminate in $P$ [10].

Postulate 4. $\text{wp}. S$ is a non-miraculous action.
Although $S$ only accesses variables in $\text{var}.F$, the action $\text{wp}.S$ is not over $\text{var}.F$ because $\text{wp}.S$ also captures the fact that other variables remain unchanged. Formally, $\text{wp}.S$ is the combination (suprema) of an action over $\text{var}.F$ with an action that does not modify the variables in $\text{var}\setminus\text{var}.F$.

**Postulate 5.** $(\forall S : S \in \text{asg}.F : \text{wp}.S = (\text{wp}.S/\text{var}.F) \cup \llbracket \text{var}\setminus\text{var}.F \rrbracket)$.

Every UNITY program $F$ can be mapped to an open system $\overline{F}$. Of course $e.\overline{F} = \text{random}$ if $F$ has no local variables because a UNITY program do not restrict its environment from modifying its shared variables.

**Definition 6.** $\text{I.}\overline{F} = \text{init}.F$, $\text{Co.}\overline{F} = \{\text{wp}.S| S \in \text{asg}.F\}$, $e.\overline{F} = \llbracket \text{loc}.F\rrbracket$, $\text{Fa.}\overline{F} = \{\text{wp}.S| S \in \text{asg}.F\}$.

Definition 6 is motivated by the computational model of UNITY programs. Computation of UNITY programs start in some initial state and proceed by repeatedly executing one of the assignment statements ad infinitum. The choice of the next statement to be executed is nondeterministic but each statement must be selected infinitely often. When viewing programs as open systems, the interference of the environment must be taken into account: program steps may be interleaved with arbitrary environment steps that may change the values of all variables except for the local ones. For instance, a computation of program $F$ in Fig. 1 might start as follows:

$$(x, y \mapsto 2, 1) \xrightarrow{c} (x, y \mapsto 4, 2) \xrightarrow{\epsilon} (x, y \mapsto 0, 2) \xrightarrow{c} (x, y \mapsto -1, 3) \xrightarrow{c}.$$ 

4. Properties

This section introduces invariant, next-state, and progress properties of open systems, hence of UNITY programs. Strongest invariants are given a prominent role in regard to their importance in the forthcoming analysis of parallel composition.

4.1. Strongest invariant

The predicate $\text{si}.O$ is the strongest invariant of $O$: $\text{si}.O.\sigma$ exactly when $\sigma$ is reachable in $O$. This predicate corresponds to Lamport's $\text{sin}(\Delta, Q)$ [17] where $Q$ is $\text{I.}O$ and the set $\Delta$ of actions is $\text{Co.}O \cup \{e.O\}$.

**Definition 7.** $\text{si}.O$ is the smallest (w.r.t. $\subseteq$) solution $X$ of

$$(\text{I.}O \subseteq X) \land (X \subseteq (e.O \cap e.O).X).$$

Let $a^\ast$ be defined by $a^\ast.P.\tau \equiv (\exists \sigma : \sigma[a] : \tau : P.\sigma)$. Then, $P \subseteq a.Q$ is equivalent to $a^\ast.P \subseteq Q$. This relation between $a$ and $a^\ast$, the monotonicity of $a^\ast$, and the Knaster–Tarski theorem ensure the existence of a strongest solution in Definition 7. An
equivalent but less workable definition of $\text{si}.O$ is

$$\text{si}.O = \langle \bigcap X : X = \text{I}.O \cup (e.O \cap c.O)^*.X : X \rangle.$$ 

First, Theorem 8 asserts that extending the set of initial states, or allowing less deterministic actions, increases the set of reachable states. Its short calculational proof illustrates the power gained by abstracting from computations to reason with predicate transformers and extreme solutions of equations in predicates. Next, Theorem 9 asserts that all states become reachable if the action $\text{random}$ is allowed.

**Theorem 8.** $(\text{I}.O \subseteq \text{I}.O') \land (e.O' \cap c.O' \subseteq e.O \cap c.O) \Rightarrow (\text{si}.O \subseteq \text{si}.O').$

**Proof.** Assume the left-hand side, and let $X$ be a solution of the equation defining $\text{si}.O'$:

$$(\text{I}.O' \subseteq X) \land (X \subseteq (e.O' \cap c.O').X)$$

$$\Rightarrow \{ \text{LHS, Definition of } \subseteq \}$$

$$(\text{I}.O \subseteq X) \land (X \subseteq (e.O \cap e.O).X)$$

$$\Rightarrow \{ \text{Definition 7} \}$$

$$\text{si}.O \subseteq X. \quad \square$$

**Theorem 9.** $(e.O \cap c.O \subseteq \text{random}) \land (\text{I}.O \neq \text{False}) \Rightarrow (\text{si}.O = \text{True}).$

### 4.2. Axiomatic semantics of properties

System $O$ has the property 'invariant $P$' if $P.\sigma$ for all reachable states $\sigma$ of $O$. Adopting the convenient notation of [6], properties are viewed as functions from systems to booleans.

**Definition 10.** (invariant $P$).$O \equiv \text{si}.O \subseteq P.$

Let $g$ be a reflexive action. System $O$ has the property 'guarantee $g$' if the relation $[g]$ holds for every component step, i.e. if $\sigma_{k-1}[g] \sigma_k$ whenever $l_k = c$. System $O$ certainly has that property if all actions of $O$ refine $g$ but this condition is stronger than necessary. Indeed, steps from a component are from reachable states only and therefore $c.O$ must be restricted to $\text{si}.O$.

**Definition 11.** (guarantee $g$).$O \equiv g \sqsubseteq c.O \sqsubseteq \text{si}.O.$

System $O$ has the property $P$ leadsto $Q$ if every computation in $O$ has a $Q$ state beyond every $P$ state, i.e.

$$\forall k : k \geq 0 : P.\sigma_k \Rightarrow (\exists j : j \geq k : Q\sigma_j).$$

This property is defined in terms of the predicate transformer $\text{wlt}.O$ (weakest leadsto); any sequence of states starting in $\text{wlt}.O.Q$ and obtained by executing environment
or component actions must have an intermediate state in $Q$ if each action in $\text{Fa}.O$ is executed infinitely often. The transformer $\text{wlt}.O$ is defined from the transformer $\text{stp}.O.a$ where $\text{stp}.O.a.P$ requires $P$ to be reached in zero steps, or via a sequence of actions followed by $a$. Both transformers are generalised from [13]. Note that $\text{wlt}.O$ is not an action because it is not finitely conjunctive [15].

**Definition 12.** $\text{wlt}.O.P$ is the strongest solution $X$ of

$$(P \subseteq X) \land ((\bigcup a : a \in \text{Fa}.O : \text{stp}.O.a.X) \subseteq X)$$

where $\text{stp}.O.a.P$ is the weakest solution $X$ of

$$(X \cap \text{PC}\subseteq (c.O \cap e.O).X) \land (X \cap \text{PC}\subseteq a.P).$$

The condition $P \subseteq \text{wlt}.O.Q$ ensures the desired property but is again stronger than necessary. Unreachable states must be discarded.

**Definition 13.** $(P \leadsto Q).O \equiv P \subseteq (\text{wlt}.O \downarrow \text{si}.O).Q.$

Invariant, next-state, and progress properties have been defined individually. Those properties can be combined by (finite or infinite) conjunction. System $O$ has the property $\langle \& i : i \in I : \Phi_i \rangle$ if it has the property $\Phi_i$ for each $i \in I$.

**Definition 14.** $\langle \& i : i \in I : \Phi_i \rangle.O \equiv \langle \forall i : i \in I : \Phi_i.O \rangle.$

*Properties of UNITY programs*: Definition 6 turns UNITY programs into open systems. Properties of program $F$ are thus defined as properties of $\overline{F}$.

**Definition 15.** $\Phi.F \equiv \Phi.\overline{F}.$

Because of arbitrary interference on the shared variables, UNITY programs enjoy few properties. Consider, for instance, the simple program $F$ that repeatedly executes the unique statement $x := x + 1; F$ does not have the property $x = 1 \leadsto x > 3$ because $x$ may be decreased by environment steps between two steps of $F$. The weakness of the properties of UNITY programs follows from the observation that the strongest invariant of a UNITY program is over its local variables.

4.3. Other properties

The following properties are not considered to be part of the specification logic. First, $\text{inv}$ properties are introduced as a reminder of the confusion about invariants in the UNITY logic, Next, $\text{unless}$ properties are introduced as an equivalent way of specifying next-state properties. Finally, $\text{ensures}$ properties are introduced to give an equivalent alternative semantics to $\text{leadsto}$ properties. In contrast with the properties defined above, $\text{inv}$ and $\text{ensures}$ properties cannot be interpreted in terms of computations.
System $O$ has the property $\text{inv } P$ if $P$ holds initially and is preserved by every action of the system.

**Definition 16.** $(\text{inv } P).O \equiv (I.O \subseteq P) \land (P \subseteq (e.O \cap c.O).P)$.

The property $\text{inv } P$ is stronger than $\text{invariant } P$ because all solutions $X$ in Definition 7 are $\text{invariant } P$ properties but not conversely. This difference, which is well documented in [14,22], has led to a great deal of confusion about the treatment of invariants in UNITY. On the one hand, $\text{inv } P$ properties compose well but invariant predicates cannot be replaced by $True$ in other properties, thus making the set of derivable properties very small. On the other hand, $\text{invariant } P$ properties of closed systems [22] (no environment action) do not compose well but invariant predicates can be substituted for $True$ in deriving other properties.

System $O$ has the property 'P unless Q' if every component step transforms $P \cap Q^c$ states into $P \cup Q$ states, i.e. if $(P \cup Q).\sigma_k$ whenever $(P \cap Q^c).\sigma_{k-1}$ and $l_k = c$ in a computation of $O$. All $\text{unless } P$ properties can be turned into equivalent $\text{guarantee } P$ properties and conversely.

**Definition 17.** $(P \text{ unless } Q).O \equiv (P \cap Q^c) \subseteq (c.O \downarrow si.O).(P \cup Q)$.

System $O$ has the property $P \text{ ensures } Q$ if all (environment and component) actions of $O$ transform $P \cap Q^c$ states into $P \cup Q$ states and there exists a fair component action that transforms $P \cap Q^c$ states into $Q$ states. Actions are restricted to $si.O$.

**Definition 18.** $(P \text{ ensures } Q).O \equiv (\exists a : a \in Fa.O : \phi)$ where

$$\phi \equiv P \cap Q^c \subseteq ((e.O \cap c.O) \downarrow si.O).(P \cup Q) \cap (a \downarrow si.O).Q.$$

Theorem 20 below gives an alternative characterisation of $\text{leadsto } P$ properties in terms of the relation $\leadsto_O$ defined from $\text{ensures } P$ properties. That theorem is a consequence of a more general theorem due to Jutla et al. [13]: the parameters of [13] can be chosen in such a way that all required conditions hold (in particular, actions must be non-miraculous). Soundness and completeness results for progress properties are based on Theorem 20.

**Definition 19.** $\leadsto_O$ is the strongest relation $\triangleright$ on predicates such that

$$(P \text{ ensures } Q).O \Rightarrow P \triangleright Q$$

$$P \triangleright X \land X \triangleright Q \Rightarrow P \triangleright Q$$

$$\langle i : i \in I : P_i \triangleright Q \rangle \Rightarrow \langle \bigcup i : i \in I : P_i \rangle \triangleright Q.$$

**Theorem 20.** $(P \text{ leadsto } Q).O \equiv P \leadsto_O Q$. 

5. Assumption-Commitment properties

This section introduces assumption-commitment properties of open systems. The rely condition in those properties is a supplementary assumption on the interference of the environment in the computation of a component. In addition to an increase in the number of properties enjoyed by open systems, this rely condition provides an elegant treatment of hiding in concurrent programs.

5.1. Axiomatic semantics

Consider again the simple program $F$ which repeatedly executes the unique statement $x:=x+1$. Under the assumption that $x$ is not decreased by environment steps, $F$ has the property $x = 1 \implies x > 3$. This assumption on environment steps is called the rely condition and is represented by a reflexive action $r$. In this example, $\sigma[r]_\tau$ if and only if $\tau(x) \geq \sigma(x)$. In a first-order language, $[r]$ can be represented by the formula $x' \geq x$ where primed variables refer to the next state.

A system has the property $(J, r, O)$ if it has the property $O$ under the rely condition $r$ and the precondition $J$; the precondition is a supplementary assumption on the initial states. This property can be easily formalised by adding the predicate $J$ to $I.O$ and the action $r$ to $e.O$ to define a new system $(J, r, O)$. The usefulness of the generalisation to open systems clearly appears in Definition 22: reasoning on assumption-commitment properties of one system (of e.g. a UNITY program) amounts to reasoning on ordinary properties of another open system (which often does not correspond to a UNITY program).


**Definition 22.** $(J, r, \Phi).O \equiv \Phi.(J, r, O)$.

In operational terms, adding the precondition $J$ and the rely condition $r$ amounts to restricting the set of computations to those starting in $J$ and for which $\sigma_{k-1}[r]_\tau \sigma_k$ whenever $l_k = e$; less computations mean more properties. The reduction of the set of reachable states is captured by Theorem 23 which follows from Theorem 8 and the observation that $\sqcup$ gives suprema.

**Theorem 23.** $si.(J, r, O) \subseteq si.O$.

Ordinary properties can of course be retrieved by choosing $J = True$ and $r = random$; all inference rules presented in the sequel remain sound if $J$ and/or $r$ are replaced everywhere by $True$ and $random$ respectively. Properties of a UNITY program $F$ extended with an initial condition $shinit.F$ on the shared variables are retrieved by fixing $J = shinit.F$; all inference rules remain sound if $J$ is fixed in that way.
5.2. Hiding

The UNITY program $F[(V, P)]$ is obtained from $F$ by turning all the shared variables in $V$ into local variables which satisfy the initial condition $P$. The corresponding open system is obtained by imposing the constraint that the variables in $V$ are not modified by environment steps.

**Definition 24.** Let $V \subseteq \text{shr}.F$ and let $P$ be over $V$. Then, $F[(V, P)]$ is defined by

$$
\text{shr.}(F[(V, P)]) = \text{shr}.F \setminus V, \quad \text{init.}(F[(V, P)]) = \text{init}.F \cap P
$$

$$
\text{loc.}(F[(V, P)]) = \text{loc}.F \cup V, \quad \text{asg.}(F[(V, P)]) = \text{asg}.F.
$$

**Definition 25.** $I.(O[(V, P)]) = I.O \cap P$, $\text{Co.}(O[(V, P)]) = \text{Co}.O$, $\text{e.}(O[(V, P)]) = e.O \sqcup [=}v]$, $\text{Fa.}(O[(V, P)]) = \text{Fa}.O$.

**Theorem 26.** $\overline{F}[(V, P)] = F[(V, P)]$.

Once rely conditions are available, coping with hiding is straightforward. Next is a sound and complete inference rule for deriving assumption-commitment properties of $O[(V, P)]$ from assumption-commitment properties of $O$.

**Rule 27.** $(J \cap P, r \sqcup [=}v], \Phi).O \equiv (J, r, \Phi).O[(V, P)]$.

5.3. Additional inference rules

The proof system for assumption-commitment properties of open systems consists of rules for hiding, rules for parallel composition, rules for elementary systems, plus a number of adaptation rules that manipulate or combine properties of the same system. The adaptation rules needed to achieve completeness and the rules for elementary systems are given next; hiding has been discussed above; parallel composition is discussed in the next section.

First, the invariant rules capture the controversial UNITY 'substitution axiom': invariants can be replaced by $\text{True}$ and conversely. The proof is by reasoning on the system $O' = (J, r, O)$ from the observation that $s_i.O'$ occurs in the definition of all properties of $O'$.

**Rule 28.** If $(J, r, \text{invariant } I).O$ then

$$
(J, r, \text{invariant } P \land I).O \equiv (J, r, \text{invariant } P).O
$$

$$
(J, r, \text{guarantee } g \uparrow I).O \equiv (J, r, \text{guarantee } g).O
$$

$$
(J, r, P \land I \text{ leadsto } Q).O \equiv (J, r, P \text{ leadsto } Q).O.
$$

Weakening rules are given next.
Rule 29. If \( J' \subseteq J, r \subseteq r', g' \subseteq g, I \subseteq I', P' \subseteq P, Q \subseteq Q' \), then

\[
(J, r, \text{invariant } I).O \Rightarrow (J', r', \text{invariant } I').O \\
(J, r, \text{guarantee } g).O \Rightarrow (J', r', \text{guarantee } g').O \\
(J, r, P \text{ leadsto } Q).O \Rightarrow (J', r', P' \text{ leadsto } Q').O.
\]

The transitivity and disjunction rules for leadsto properties are straightforward consequences of Theorem 20 for the open system \((J, r, O)\).

Rule 30. \((J, r, P \text{ leadsto } X).O \wedge (J, r, X \text{ leadsto } Q).O \Rightarrow (J, r, P \text{ leadsto } Q).O\).

Rule 31. \((\forall i : i \in I : (J, r, P_i \text{ leadsto } Q_i).O) \Rightarrow (J, r, \bigcup_i : i \in I : P_i \text{ leadsto } Q).O\).

Elementary systems consist of a single action that is executed infinitely often. All properties of elementary systems can be deduced from Rules 28, 30, 31, 33, 34, and 35.

Definition 32. \(\text{Elem}(O)\) if and only if \(I.O = \text{True}, e.O = \text{random}, \ Co.O = \{a\}\) and \(Fa.O = \{a\}\) for some non-miraculous action \(a\).

Rule 33. \(\text{Elem}(O) \land (J \subseteq P) \land (P \subseteq e.O.P) \land (P \subseteq r.P) \Rightarrow (J, r, \text{invariant } P).O\).

Rule 34. \(\text{Elem}(O) \land g \subseteq e.O \Rightarrow (J, r, \text{guarantee } g).O\).

Rule 35. \(\text{Elem}(O) \land (P \cap Q^c \subseteq e.O.Q) \land (P \cap Q^c \subseteq r.(P \cup Q)) \Rightarrow (J, r, P \text{ leadsto } Q).O\).

6. Parallel composition

Composition of UNITY programs by union is presented as a special case of a more general notion of parallel composition for open systems. The principal advantage of this generalisation is the ability to explain (the soundness and completeness of) composition rules for assumption-commitment properties by considering the parallel composition of open systems which are not UNITY programs. This section must be viewed as a stepwise presentation of composition rules for safety and progress properties, with intermediate theorems to emphasise the key steps.

6.1. Composition of open systems

Open systems are intended to represent components that can be executed in parallel with others. The parallel composition of two open systems then represents a new component whose definition captures four observations on parallel composition in an interleaving approach to concurrency. Firstly, initial states of a composite system are initial states of its components. Secondly, the environment of a composite system is
the intersection of the environments of its components. Thirdly, each action taken by a
composite system is taken by one of its components. Finally, all fair actions of every
component are executed infinitely often in the composition. The second observation is
formalised with the help of the operator $\cup$ of Section 2: $[a_1 \cup a_2]$ is the intersection
of $[a_1]$ and $[a_2]$. From the third observation, $c.(O_1 \parallel O_2) = c.O_1 \cap c.O_2$.

**Definition 36.** $I.(O_1 \parallel O_2) = I.O_1 \cap I.O_2$, $e.(O_1 \parallel O_2) = e.O_1 \cup e.O_2$, $Co.(O_1 \parallel O_2) = Co.O_1 \cup Co.O_2$, $Fa.(O_1 \parallel O_2) = Fa.O_1 \cup Fa.O_2$.

In general, properties of $O_1 \parallel O_2$ cannot be related to properties of $O_1$ and $O_2$ be-
cause Definition 36 fails to capture the observation that in a composite system each
component is part of the environment of the other. This additional observation can be
captured by the condition $O_1 \triangledown_s O_2$.

**Definition 37.** $O_1 \triangledown_s O_2 \equiv (e.O_2 \subseteq c.O_1) \land (e.O_1 \subseteq c.O_2)$.

Under the condition $O_1 \triangledown_s O_2$, every reachable state of $O_1 \parallel O_2$ is reachable in $O_1$ and
$O_2$ and therefore $O_1 \parallel O_2$ inherits all invariants of $O_1$ and $O_2$. Theorem 38 follows from
Theorem 8 by standard reasoning in a lattice.

**Theorem 38.** $O_1 \triangledown_s O_2 \Rightarrow si.(O_1 \parallel O_2) \subseteq si.O_1 \cap si.O_2$.

The condition $O_1 \triangledown_s O_2$ is stronger than necessary. The analogue of Theorem 38
(Theorem 41 below) only requires the restriction of $c.O_1$ and $c.O_2$ to reachable states
to be refinement of $e.O_2$ and $e.O_1$ respectively; this new cooperation condition
is denoted by $O_1 \triangledown_w O_2$.

**Definition 39.** $O_1 \triangledown_w O_2 \equiv (e.O_2 \subseteq c.O_1 \downarrow si.O_1) \land (e.O_1 \subseteq c.O_2 \downarrow si.O_2)$.

Observe that via the strongest invariants, $e.O_1$ is now used to conclude that $c.O_1$
is a refinement of $e.O_2$, which in turn is used to conclude that $c.O_2$ is a refine-
ment of $e.O_1$. This apparent circularity (from $e.O_1$ to $e.O_1$) is typical of assumption-
commitment reasoning (see [2] for a detailed discussion of this). The adequacy of
the cooperation condition is however justified by Theorem 41 which lies at the heart
of all composition rules. Its proof and all other proofs in this section are of inter-
est by themselves because they are entirely carried out by reasoning with predicate
transformers and extreme solutions of equations in predicates. This contrasts with the
more usual operational style of reasoning when dealing with assumption-commitment
specifications.

**Theorem 40.** Let $O = O_1 \parallel O_2$. Then,

$$(e.O_1 \subseteq c.O_2 \downarrow si.O_2) \Rightarrow (e.O_1 \cap e.O_1 \subseteq (e.O \cap e.O) \downarrow (si.O_1 \cap si.O_2))$$
$$(e.O_2 \subseteq c.O_1 \downarrow si.O_1) \Rightarrow (e.O_2 \cap e.O_2 \subseteq (e.O \cap e.O) \downarrow (si.O_1 \cap si.O_2)).$$
Proof. Let $S = \text{si}.O_1 \cap \text{si}.O_2$:

\[ e.O_1 \subseteq c.O_2 \ \text{\&} \ \text{si}.O_2 \]

$\Rightarrow$ \{Definition of infima and suprema\}

\[ e.O_1 \cap c.O_1 \subseteq (e.O_1 \cup e.O_2) \cap c.O_1 \cap c.O_2 \ \text{\&} \ \text{si}.O_2 \]

$\Rightarrow$ \{\text{if } a \subseteq a \downarrow P \text{ and } a \downarrow P \text{ is antimonotonic in } P\}

\[ e.O_1 \cap c.O_1 \subseteq (e.O_1 \cup e.O_2) \downarrow S \cap c.O_1 \downarrow S \cap c.O_2 \downarrow S \]

$\equiv$ \{Definition of infima and suprema\}

\[ e.O_1 \cap c.O_1 \subseteq ((e.O_1 \cup e.O_2) \cap c.O_1 \cap c.O_2) \downarrow S \]

$\equiv$ \{Definition 36\}

\[ e.O_1 \cap c.O_1 \subseteq (e.O \cap c.O) \downarrow S. \quad \Box \]

Theorem 41. $O_1 \nabla_w O_2 \Rightarrow (\text{si}.O_1 \| O_2) \subseteq \text{si}.O_1 \cap \text{si}.O_2$.

Proof. Let $O = O_1 \| O_2$, $S_1 = \text{si}.O_1$, $S_2 = \text{si}.O_2$. Assume $O_1 \nabla_w O_2$:

\[ \text{true} \]

$\equiv$ \{Definitions 7 and 36\}

\[ (I.O \subseteq S_1 \cap S_2) \wedge (S_1 \subseteq (e.O \cap c.O) \cdot S_1) \wedge (S_2 \subseteq (e.O \cap c.O) \cdot S_2) \]

$\Rightarrow$ \{Theorem 40\}

\[ (I.O \subseteq S_1 \cap S_2) \wedge (S_1 \subseteq ((e.O \cap c.O) \downarrow (S_1 \cap S_2)) \cdot S_1) \wedge (S_2 \subseteq ((e.O \cap c.O) \downarrow (S_1 \cap S_2)) \cdot S_2) \]

$\Rightarrow$ \{\text{if } a \subseteq a \downarrow P \text{ and } a \downarrow P \text{ is antimonotonic in } P\}

\[ (I.O \subseteq S_1 \cap S_2) \wedge (S_1 \cap S_2 \subseteq (e.O \cap c.O) \cdot S_1 \cap (e.O \cap c.O) \cdot S_2) \]

$\equiv$ \{Actions are universally conjunctive\}

\[ (I.O \subseteq S_1 \cap S_2) \wedge (S_1 \cap S_2 \subseteq (e.O \cap c.O) \cdot (S_1 \cap S_2)) \]

$\Rightarrow$ \{Definition 7\}

\[ \text{si}.O \subseteq S_1 \cap S_2. \quad \Box \]

The inheritance of \textit{invariant} properties and the composition of \textit{guarantee} properties are immediate consequences of Theorem 41.

Theorem 42. $O_1 \nabla_w O_2 \wedge (\text{invariant } P).O_1 \Rightarrow (\text{invariant } P). (O_1 \| O_2)$.

Theorem 43. $O_1 \nabla_w O_2 \wedge (\text{guarantee } g_1).O_1 \wedge (\text{guarantee } g_2).O_2 \Rightarrow (\text{guarantee } g_1 \cap g_2). (O_1 \| O_2)$.
An equally interesting and perhaps more surprising consequence of Theorem 41 is the inheritance of progress properties by composition. Theorem 45 follows from Theorems 20 and 44.

**Theorem 44.** \( O_1 \land_w O_2 \land (P \text{ ensures } Q), O_1 \Rightarrow (P \text{ ensures } Q), (O_1 || O_2) \).

**Proof.** Let \( O = O_1 || O_2 \). Assume \( O_1 \land_w O_2 \). The proof follows from Definition 18, \( Fa. O = Fa. O_1 \cup Fa. O_2 \), and

\[
(P \cap Q^c \subseteq ((e. O_1 \cap c. O_1) \downarrow si. O_1) \cdot (P \cup Q) \cap (a \downarrow si. O_1) \cdot Q)
\]

\[
\Rightarrow \{ \text{Theorem 40 and } P_1 \subseteq (a \downarrow P_2), P_3 \equiv P_1 \cap P_2 \subseteq a. P_3 \}
\]

\[
(P \cap Q^c \cap si. O_1 \cap si. O_2 \subseteq (e. O \cap c. O) \cdot (P \cup Q) \cap a. Q)
\]

\[
\Rightarrow \{ \text{Theorem 41} \}
\]

\[
(P \cap Q^c \cap si. O \subseteq (e. O \cap c. O) \cdot (P \cup Q) \cap a. Q)
\]

\[
\equiv \{ P_1 \subseteq (a \downarrow P_2), P_3 \equiv P_1 \cap P_2 \subseteq a. P_3 \}
\]

\[
(P \cap Q^c \subseteq ((e. O \cap c. O) \downarrow si. O) \cdot (P \cup Q) \cap (a \downarrow si. O) \cdot Q). \quad \Box
\]

**Theorem 45.** \( O_1 \land_w O_2 \land (P \text{ leadsto } Q), O_1 \Rightarrow (P \text{ leadsto } Q), (O_1 || O_2) \).

6.2. Composition of UNITY programs

Composition of UNITY programs by union merely consists of taking the union of their sets of assignment statements. Composition is however restricted to programs which do not access each other's local variables; of course, local variables can be renamed first. For example, program \( F \) in Fig. 1 is the union of programs \( G \) and \( H \) in Fig. 2.

**Definition 46.** Let \( F_1, F_2 \) be such that no variable of one program is a local variable of the other. Then, \( F_1 || F_2 \) is defined by

\[
\text{shr.}(F_1 || F_2) = \text{shr.} F_1 \cup \text{shr.} F_2, \quad \text{l}.(F_1 || F_2) = \text{l}. F_1 \cap \text{l}. F_2,
\]

\[
\text{loc.}(F_1 || F_2) = \text{loc.} F_1 \cup \text{loc.} F_2, \quad \text{asg.}(F_1 || F_2) = \text{asg.} F_1 \cup \text{asg.} F_2.
\]
As expected, the union of UNITY programs corresponds to the parallel composition of their induced open systems. Interestingly, the cooperation condition always holds when UNITY programs are composed.

**Theorem 47.** \( F = F_1 \parallel F_2 \Rightarrow F_1 \cup F_2 \wedge F = F_1 \parallel F_2. \)

**Proof (sketch).** \( F = F_1 \parallel F_2 \) follows from \([-r_{11} \cup r_{22} = -r_1 \cup -r_2]. \) Then, \( \text{var}.F_2 \cap \text{loc}.F_1 = \emptyset \) and Postulate 5 implies

\[
(\forall S : S \in \text{assign}.F_2 : [=\text{loc}.F_1] \subseteq \text{wp}.S)
\]

which is equivalent to \((e.F \subseteq c.F_2). \)

Theorems 42, 45, and 47 suggest that all invariant and leadsto properties of \( F_1 \) and \( F_2 \) are inherited by \( F_1 \parallel F_2. \) That is true. However, as discussed in Section 4, the only invariants of \( F_1 \) and \( F_2 \) are local invariants and therefore only weak properties are inherited by composition. A more interesting consequence of Theorems 41 and 47 is the ability to deduce stronger properties of a composite program from assumption-commitment properties of its components.

**6.3. Composition of assumption-commitment properties**

Instead of using Theorem 41 with \( O_1 \) and \( O_2, \) the key idea is to use that theorem with \((J_1, r_1, O_1)\) and \((J_2, r_2, O_2)\) and thus deduce properties of \( O_1 \parallel O_2 \) from properties of \( O_1 \) and \( O_2 \) that hold under the rely conditions \( r_1 \) and \( r_2, \) respectively.

The cooperation condition \((J_1, r_1, O_1) \cup (J_2, r_2, O_2)\) requires the cooperation condition \( O_1 \cup O_2 \) plus the conditions

\[
\begin{align*}
    r_1 &\subseteq c.(J_2, r_2, O_2) \parallel si.(J_2, r_2, O_2), \\
    r_2 &\subseteq c.(J_1, r_1, O_1) \parallel si.(J_1, r_1, O_1).
\end{align*}
\]

These additional conditions follow from \( r_1 \subseteq g_2 \) and \( r_2 \subseteq g_1 \) if guarantee \( g_1 \) and guarantee \( g_2 \) are properties of \((J_1, r_1, O_1)\) and \((J_2, r_2, O_2)\) respectively. The presence of the strongest invariants in the semantics of guarantee properties explains why the condition in Definition 37 is too strong.

**Theorem 48.** \( O_1 \cup O_2 \wedge (r_1 \subseteq g_2) \wedge (r_2 \subseteq g_1) \wedge (J_1, r_1, O_1) \parallel (J_2, r_2, O_2) \Rightarrow (J_1, r_1, O_1) \cup (J_2, r_2, O_2). \)

The composition rules are now formulated. By Definition 22, Theorem 48, and Theorem 49, Rules 50, 51, and 52 are direct consequences of Theorems 42, 43, and 45, respectively. The roles of \( O_1 \) and \( O_2 \) in Rules 50 and 52 can of course be interchanged.

**Theorem 49.** \( (J_1 \cap J_2, r_1 \cup r_2, O_1 \cup O_2) = (J_1, r_1, O_1) \parallel (J_2, r_2, O_2). \)
Rule 50. \( O_1 \wedge (r_1 \subseteq g_2) \wedge (r_2 \subseteq g_1) \wedge (J_1, r_1, \text{guarantee } g_1).O_1 \wedge (J_2, r_2, \text{guarantee } g_2).O_2 \wedge (J_1, r_1, \text{invariant } P).O_1 \Rightarrow (J_1 \cap J_2, r_1 \cup r_2, \text{invariant } P).(O_1 \| O_2). \)

Rule 51. \( O_1 \wedge (r_1 \subseteq g_2) \wedge (r_2 \subseteq g_1) \wedge (J_1, r_1, \text{guarantee } g_1).O_1 \wedge (J_2, r_2, \text{guarantee } g_2).O_2 \wedge (J_1, r_1, \text{guarantee } u_1).O_1 \wedge (J_2, r_2, \text{guarantee } u_2).O_2 \Rightarrow (J_1 \cap J_2, r_1 \cup r_2, \text{guarantee } u_1 \cap u_2).(O_1 \| O_2). \)

Rule 52. \( O_1 \wedge (r_1 \subseteq g_2) \wedge (r_2 \subseteq g_1) \wedge (J_1, r_1, \text{guarantee } g_1).O_1 \wedge (J_2, r_2, \text{guarantee } g_2).O_2 \wedge (J_1, r_1, P \text{ leadsto } Q).O_1 \Rightarrow (J_1 \cap J_2, r_1 \cup r_2, P \text{ leadsto } Q).(O_1 \| O_2). \)

By Theorem 47, the premise \( O_1 \wedge O_2 \) disappears in case \( O_1 = \overline{F}_1 \) and \( O_2 = \overline{F}_2 \) for UNITY programs \( F_1 \) and \( F_2 \).

6.4. Completeness

It is next shown that, under certain conditions, every assumption-commitment property \( (J, r, \Phi) \) of \( O_1 \| O_2 \) can be deduced from assumption-commitment properties of \( O_1 \) and \( O_2 \). Let

\[
J_1 = J \cap 1.O_2, \quad g_1 = c.O_1 \cup \text{skip}, \quad r_1 = (r \cup e.O_2) \cap g_2,
\]

\[
J_2 = J \cap 1.O_1, \quad g_2 = c.O_2 \cup \text{skip}, \quad r_2 = (r \cup e.O_1) \cap g_1.
\]

Since \( \cap \) gives infima, \( r_1 \subseteq g_2, \ r_2 \subseteq g_1 \), and

\[
(J_1, r_1, \text{guarantee } g_1).O_1 \wedge (J_2, r_2, \text{guarantee } g_2).O_2.
\]

Next, by elementary reasoning in a lattice:

\[
\text{I}.(J_1, r_1, O_1) = \text{I}.(J, r, O_1 \| O_2)
\]

\[
\text{I}.(J_2, r_2, O_2) = \text{I}.(J, r, O_1 \| O_2)
\]

\[
e.(J, r, O_1 \| O_2) \cap c.(J, r, O_1 \| O_2) \subseteq e.(J_1, r_1, O_1) \cap c.(J_1, r_1, O_1)
\]

\[
e.(J, r, O_1 \| O_2) \cap c.(J, r, O_1 \| O_2) \subseteq e.(J_2, r_2, O_2) \cap c.(J_2, r_2, O_2),
\]

which by Theorem 8 implies

\[
(s \text{I}.(J_1, r_1, O_1) \subseteq s \text{I}.(J, r, O_1 \| O_2)) \wedge (s \text{I}.(J_2, r_2, O_2) \subseteq s \text{I}.(J, r, O_1 \| O_2)).
\]

Consequently, by Definition 22,

\[
(J, r, \text{invariant } P).(O_1 \| O_2) \Rightarrow (J_1, r_1, \text{invariant } P).O_1
\]

\[
(J, r, \text{guarantee } u).(O_1 \| O_2) \Rightarrow (J_1, r_1, \text{guarantee } u).O_1
\]

\[
(J, r, \text{guarantee } u).(O_1 \| O_2) \Rightarrow (J_2, r_2, \text{guarantee } u).O_2,
\]

and, from \( \text{Fa}.(O_1 \| O_2) = \text{Fa}.O_1 \cup \text{Fa}.O_2 \),

\[
(J, r, P \text{ ensures } Q).(O_1 \| O_2) \Rightarrow (J, r, P \text{ ensures } Q).O_1 \lor (J, r, P \text{ ensures } Q).O_2.
\]
Therefore, if \((J, r, \Phi)\) is a property of \(O_1 || O_2\), the property \((J_1 \cap J_2, r_1 \cup r_2, \Phi)\) of \(O_1 || O_2\) can be deduced from properties of \(O_1\) and \(O_2\) by application of the composition rules if \(O_1 \nabla_w O_2\) holds. The property \((J \cap I.(O_1 || O_2), r \sqcup e.(O_1 || O_2), \Phi)\) can then be deduced by application of the weakening rules. In case \(\Phi\) is a \textit{leadsto} property, the transitivity and disjunction rules must be applied after the composition rule (Theorem 20).

This leads to Theorem 53. The restrictions on the completeness result are further discussed in Section 7 and then lifted in Section 8.

**Theorem 53.** If \(O_1 \nabla_w O_2, I.(O_1 || O_2) = \text{True}\) and \(e.(O_1 || O_2) = \text{random}\), then for every property \((J, r, \Phi)\) of \(O_1 || O_2\) there exists sets \((J_1^i, r_1^i, \Phi_i^1)\) and \((J_2^i, r_2^i, \Phi_i^2)\) of properties of, respectively, \(O_1\) and \(O_2\) from which \((J, r, \Phi).(O_1 || O_2)\) can be deduced by application of the composition, weakening, transitivity, and disjunction rules.

### 7. Compositionality

In general, a component consists of external and internal features. In case of a UNITY program \(F\), \(\text{shr}.F\) is an interface feature, whereas \(\text{loc}.F, I.F,\) and \(\text{asg}.F\) are internal features. To separate compositionality issues from the fundamental results on composition, that distinction has been intentionally overlooked in previous sections.

In brief, a compositional approach to the development and/or verification of concurrent programs is one that never refers to the internal features of a component. This has two implications. Firstly, the encapsulation principle [21] requires that properties of a component should be over its shared variables only, so that components with the same observable properties are interchangeable even if their implementation differ. Secondly, the properties of a composite system should be verifiable from the properties of its components without knowledge of their interior structure; this is crucial if a design decision to decompose a specification of a system into specifications of its components has to be justified before the components are further developed.

This section first introduces observable properties, then looks at the compositionality of the proof system, and finally discusses the impact of observability on the completeness results; detailed formal statements of the claims can be found in [8]. Although the generalization to open systems has been most useful in previous sections, this section deals with UNITY programs only. The cooperation condition \(O_1 \nabla_w O_2\), whose role is essential when discussing compositionality, always holds if \(O_1 = \overline{F_1}\) and \(O_2 = \overline{F_2}\) but this is not the case for arbitrary open systems \(O_1\) and \(O_2\).

**Definition 54.** Let \(V \subseteq \text{VAR}\). The properties \textit{invariant} \(P\), \textit{guarantee} \(g\), and \(P\) \textit{leadsto} \(Q\) are over \(V\) if \(P, Q,\) and \(g\) are over \(V\); \(\langle \&i : i \in I : \Phi_i \rangle\) is over \(V\) if \(\Phi_i\) is over \(V\) for each \(i \in I; (J, r, \Phi)\) is over \(V\) if \(\Phi, J,\) and \(r\) are over \(V\).

**Definition 55.** \(\Phi\) is an observable property of \(F\) if \(\Phi\) is over \(\text{shr}.F\).
Ideally, applying an inference rule with observable properties in the premises should yield observable properties in the conclusion. This can be easily achieved by imposing that \( J \) and \( r \) be over \( \text{shr}.F \setminus V \) in Rule 27, and \( J',r',g',I',P',Q' \) be over \( \text{shr}.F \) in Rule 29.

### 7.1. Compositionality of the proof system

A proof system is compositional if each composition constructor (hiding and union) is matched by a corresponding inference rule and none of the rules refer to the internal features of a component [26]. For each composite program \( F = \varphi(F_1, \ldots, F_n, \alpha) \) (\( n \geq 1 \)) where \( \alpha \) is a parameter (e.g. \( \alpha = (V,P) \) for hiding), there must be a rule whose conclusion is a property \( \Phi_0 \) of \( F \) and whose only premises are properties \( \Phi_i \) of \( F_1, \ldots, F_n \), and conditions (e.g. \( r_1 \subseteq g_2 \)) relating \( \alpha, \Phi_0, \Phi_1, \ldots, \Phi_n, \text{shr}.F, \text{shr}.F_1, \ldots, \text{shr}.F_n \) only. According to that definition, the proof system for UNITY programs is compositional because the condition \( O_1 \lor O_2 \) (which refers to internal features) can be removed from the parallel composition rules (Theorem 47 for UNITY programs).

### 7.2. Compositional completeness

The proof system for UNITY is compositionally complete if it is complete for elementary programs (no local variables, unique assignment statement), if every observable property of \( F | (V,C) \) can be deduced from properties of \( F \), and if every observable property of \( F_1 | F_2 \) can be deduced from observable properties of \( F_1 \) and \( F_2 \). A complete set of rules for elementary programs has been given in Section 5. Completeness w.r.t hiding follows from Rule 27 (note that \( V \) is a subset of \( \text{shr}.F \) and \( P \) is over \( V \), hence over \( \text{shr}.F \)). Completeness w.r.t. parallel composition has been discussed in Section 6: under the stated conditions every property \((J,r,\Phi)\) of \( F_1 | F_2 \) can be proved from properties of \( F_1 \) and \( F_2 \). Unfortunately, the latter are not necessarily observable properties of \( F_1 \) and \( F_2 \), even if \((J,r,\Phi)\) is an observable property of \( F_1 | F_2 \). For instance, the action \( r_2 \) constructed in Section 6 is not over \( \text{shr}.F_2 \) because \( r \) is over \( (\text{shr}.F_1 \cup \text{shr}.F_2) \) and \( e.F_1 = \text{loc}.F \) is over \( \text{loc}.F_1 \).

Compositional completeness w.r.t. parallel composition can however be established under the condition

\[
(\star) \quad (\text{loc}.F_1 = \emptyset) \land (\text{loc}.F_2 = \emptyset) \land (\text{shr}.F_1 = \text{shr}.F_2)
\]

by choosing

\[
J_1 = J, \quad g_1 = c.F_1/\text{shr}.F_1 \cap \text{skip}, \quad r_1 = r \cap g_2,
\]

\[
J_2 = J, \quad g_2 = c.F_2/\text{shr}.F_2 \cap \text{skip}, \quad r_2 = r \cap g_1.
\]

Because \( 1.F = \text{True} \) and \( e.F = \text{random} \) if \( \text{loc}.F = \emptyset \), the condition \((\star)\) implies the conditions required in Theorem 53. The modification to the construction of \( g_1 \) and \( g_2 \).
does not affect the proof sketched in Section 6 because $O_i^*$ $(i = 1, 2)$ and $F_i$ enjoy the same properties over $\text{shr}.F_i$ if $O_i^*$ is constructed from $\overline{F_i}$ by replacing every component action $a$ of $\overline{F_i}$ with $a/\text{shr}.F_i$.

It should finally be made clear that the applicability of the composition rules is not restricted to programs without local variables and with identical sets of shared variables. The condition $(\dagger)$ is a restriction on the completeness result only. It will be lifted in Section 8 by considering specifications with auxiliary variables.

8. Auxiliary variables

The specification logic is based on a small set of temporal operators and therefore has a limited expressive power. It essentially misses some means to refer to the past behavior of a program, i.e. to carry some history information. The classical example is a buffer program $B$ that interacts with its environment via the shared variables $\text{in}$ and $\text{out}$. Typically, $B$ has the property that no data is ever lost or created. This property cannot be expressed as a property over $\text{in}$ and $\text{out}$ only because it has to refer to the sequence of data which have been received so far but which have not been transmitted yet. Of course, this property can be expressed by using a richer temporal logic but that option is rejected because it considerably raises the complexity of reasoning with the logic. Instead, this property is expressed with the help of an auxiliary variable $b$ that represents the sequence of data currently stored in the buffer. The specified property $\Phi$ is

$$(J, \{=\{b\}\}, \text{guarantee } g)$$

where $J.\sigma \equiv \sigma.b = \varepsilon$, $\sigma[g]_\tau \equiv \sigma.\text{out} \circ \sigma.b \circ \sigma.\text{in} = \tau.\text{out} \circ \tau.b \circ \tau.\text{in}$, $\varepsilon$ is the empty value, and $\circ$ denotes concatenation; the rely condition asserts that $b$ is not modified by environment actions; the guarantee property asserts that data can only be moved 'inside' the sequence $\text{out} \circ b \circ \text{in}$. Consider the program $B$ in Fig. 3 obtained from the parallel composition of two smaller buffers and let $X.\Phi$ indicate that $\Phi$ is evaluated w.r.t. a set $X$ of auxiliary variables. Then, $B$ has the property $\{b\}.\Phi$ because there exists an augmentation of $B$ with assignments to $b$ (e.g. program $B^*$) which has the property $\Phi$.

8.1. Axiomatic semantics

The following semantics of specifications with auxiliary variables is borrowed from [24]. It is required that the new value of $x \in X$ in an augmentation of $F$ depends on the previous value of $x$ and on the variables in $\text{var}.F$ only.

---

\(^2\)Notations in this paper have been chosen to ease the theoretical analysis of specifications at the semantic level; a concrete syntax should of course provide user-friendlier notations.
Program $B$

Shared $in, out : data$
Local $mid : data, s_1, s_2 : seq$ of $data$
Initially $s_1 = c \wedge s_2 = c \wedge mid = c$
Assign

\[\begin{align*}
in, s_1 & := c, s_1 \circ in \quad \text{if } in \neq c \\
mid, s_1 & := \text{hd}(s_1), \text{tail}(s_1) \quad \text{if } mid = \varepsilon \wedge s_1 \neq \varepsilon \\
\mid, s_2 & := \varepsilon, s_2 \circ mid \quad \text{if } mid \neq \varepsilon \\
\text{out}, s_2 & := \text{hd}(s_2), \text{tail}(s_2) \quad \text{if } \text{out} = \varepsilon \wedge s_2 \neq \varepsilon
\end{align*}\]

Program $B^*$

Shared $in, out : data, b : seq$ of $data$
Local $mid : data, \cdot s_1, s_2 : seq$ of $data$
Initially $s_1 = \varepsilon \wedge s_2 = \varepsilon \wedge mid = \varepsilon$
Assign

\[\begin{align*}
in, s_1, b & := \varepsilon, s_1 \circ in, b \circ in \quad \text{if } in \neq \varepsilon \\
\mid, s_1 & := \text{hd}(s_1), \text{tail}(s_1) \quad \text{if } mid = \varepsilon \wedge s_1 \neq \varepsilon \\
\mid, s_2 & := \varepsilon, s_2 \circ mid \quad \text{if } mid \neq \varepsilon \\
\text{out}, s_2, b & := \text{hd}(s_2), \text{tail}(s_2), \text{tail}(b) \quad \text{if } \text{out} = \varepsilon \wedge s_2 \neq \varepsilon
\end{align*}\]

Fig. 3. Augmented programs.

Definition 56. Statement $S$ is an $X$-extension of statement $S'$ of $F$ if an only if $S'$ is obtained from $S$ by removing the assignments to the variables in $X$ and the right-hand side of an assignment to $x \in X$ is an expression over $\text{var.} F \cup \{x\}$.

Definition 57. $G$ is an $X$-extension of $F$ if $X \cap \text{var.} F = \emptyset$, $\text{shr.} G = \text{shr.} F \cup X$, $\text{loc.} G = \text{loc.} F$, $I.G = I.F$, and there exists a bijection $m$ from $\text{asg.} G$ to $\text{asg.} F$ such that $S$ is an $X$-extension of $m(S)$ for all $S \in \text{asg.} G$.

The property $X.\Phi$ is an observable property of $F$ if $\Phi$ is over $\text{shr.} F \cup X$. In a compositional approach, the set $\text{shr.} F$ is known, and auxiliary variables can always be renamed to ensure $X \cap \text{shr.} F = \emptyset$. To cope with the case $X \cap \text{loc.} F \neq \emptyset$, a renaming of the local variables of $F$ has been allowed in Definition 58.

Definition 58. $G \succ_X F$ if and only if there exists a program $F'$ obtained from $F$ by renaming its local variables such that $G$ is an $X$-extension of $F'$.

The semantics of properties with auxiliary variables is defined next.

Definition 59. $X.\Phi.F \equiv (\exists G : G \succ_X F : \Phi.G)$. 
Remark. Internal auxiliary variables are those required to be preserved by environment actions in the rely condition; a typical example is \( b \) above. It should be clear from Program \( B^* \) in Fig. 3 that \( b = s_2 \circ mid \circ s_1 \): there exists an abstraction function from the variables of \( B \) to the internal auxiliary variable \( b \). This suggests an alternative semantics where the existential quantification over augmented programs is replaced by an existential quantification over abstraction functions (defined from the program augmented with history and prophecy variables \([l]\)). The main advantage of that alternative semantics would be its ability to cope with adaptation rules that allow internal auxiliary variables to be data-refined; it would also ease the transition with the program refinement techniques of [25] (see Section 9). The choice of the semantics in Definition 59 is perhaps a shortcoming of the approach but the completeness proof requires auxiliary variables which are not internal ones (see example with \( x := x + 1 \) and \( y := y - 1 \) below).

8.2. Interference rules

Reasoning about properties with auxiliary variables amounts to adding a layer on top of the proof system presented in previous sections. In practice, proofs can be carried out simply by assuming that the auxiliary variables are shared variables. This clearly appears in the consequence, hiding, and union rules below.

**Rule 60.** Let \( \varphi \equiv \text{shr}.G = \text{shr}.F \cup X \) in

\[
\langle \forall G : \varphi : \Phi,G \Rightarrow \Phi',G \rangle \Rightarrow (X . \Phi,F \Rightarrow X . \Phi',F).
\]

**Rule 61.** Let \( \varphi \equiv \text{shr}.G = \text{shr}.F \cup X \) in

\[
\langle \forall G : \varphi : \Phi,G \Rightarrow \Phi',(G[(V,P)]) \Rightarrow (X . \Phi,F \Rightarrow X . \Phi',(F[(V,P)])).
\]

**Rule 62.** Let \( \varphi \equiv (\text{shr}.G_1 = \text{shr}.F_1 \cup X) \land (\text{shr}.G_2 = \text{shr}.F_2 \cup X) \) in

\[
\langle \forall G_1, G_2 : \varphi : \Phi_1.G_1 \land \Phi_2.G_2 \Rightarrow \Phi.(G_1|G_2) \rangle \\
\Rightarrow (X_1 . \Phi_1.F_1 \land X_2 . \Phi_2.F_2 \Rightarrow (X_1 \cup X_2). \Phi.(F_1|F_2)).
\]

Next, consider the programs \( F \) and \( G \) whose only shared variables are \( x \) and \( y \), respectively and whose only statements are \( x := x - 1 \) and \( y := y + 1 \), respectively. Then, \( x \leq y \) is invariant in \( F|G \) if \( x = y \) initially and neither \( x \) nor \( y \) are modified by other programs. This property cannot be proved from properties of \( F \) over \( \{x\} \) and properties of \( G \) over \( \{y\} \). It can however be proved from \( \{y\}.\Phi_F.F \) and \( \{x\}.\Phi_G.G \) where

\[
\Phi_F = (\text{guarantee } [\neg y]) \land (J, r_y \cup [\neg x], \text{invariant } P) \\
\Phi_G = (\text{guarantee } [\neg x]) \land (J, r_x \cup [\neg y], \text{invariant } P) \\
J . \sigma \equiv \sigma . x = \sigma . y, \quad \sigma | r_y | \tau \equiv \tau . y \geq \sigma . y, \\
P . \sigma \equiv \sigma . x \leq \sigma . y, \quad \sigma | r_x | \tau \equiv \tau . x \leq \sigma . x.
\]
The variables $y$ and $x$ are thus auxiliary in the specifications of $F$ and $G$ respectively; of course, $x$ and $y$ are not auxiliary in the specification of $F|G$. This example illustrates why the completeness of the proof system needs Rule 63; Rule 62 is retrieved by imposing $X_1 \cap \text{shr}.F_2 = \emptyset$ and $X_2 \cap \text{shr}.F_1 = \emptyset$.

**Rule 63.** Let $\phi \equiv (\text{shr}.G_1 = \text{shr}.F_1 \cup X) \land (\text{shr}.G_2 = \text{shr}.F_2 \cup X) \land \phi'_1 = (\text{guarantee} \ [\ = X_1 \cap \text{shr}.F_1 ]) \land \phi_1 \land \phi'_2 = (\text{guarantee} \ [\ = X_1 \cap \text{shr}.F_1 ]) \land \phi_2 \land X = (X_1 \setminus \text{shr}.F_2) \cup (X_2 \setminus \text{shr}.F_1)$ in

\[
\langle \forall G_1, G_2 : \phi : \phi'_1, G_1 \land \phi'_2, G_2 \Rightarrow \phi, (G_1 \parallel[G_2]) \rangle
\]

\[
\Rightarrow (X_1 \cdot \phi'_1, F_1 \land X_2 \cdot \phi'_2, F_2 \Rightarrow X \cdot \phi, (F_1 \parallel F_2)).
\]

Finally, Rule 64 is used in the completeness proof to eliminate auxiliary variables.

**Rule 64.** If $X \cap Y = \emptyset$, $\Phi_i$, $J_i$, and $r_i$ are auxiliary variables for each $i \in I$, $J_Y$ and $r_Y$ are auxiliary variables for $Y$, $J_Y \neq False$, and $r_Y$ is non-miraculous, then

\[
\langle X \cup Y \cdot \langle \& i : i \in I : (J_i \cap J_Y, r_i \cup r_Y, \Phi_i) \rangle \cdot F \rangle
\]

\[
\Rightarrow X \cdot \langle \& i : i \in I : (J_i, r_i, \Phi_i) \rangle \cdot F.
\]

**Completeness.** Compositional completeness for properties with auxiliary variables has been proved in [8] by reusing the completeness result of Section 7. Essentially, the restriction (*)& is lifted by introducing internal auxiliary variables that mimic the local variables and auxiliary variables that represent the shared variables of another program ($x$ and $y$ in the example above).

9. **UNITY logics**

This paper’s version of the UNITY logic is now compared with others [5,6,11,22]. It is first argued that the addition of rely conditions overcomes a number of limitations of the previous variants of the logic. The complementarity with Udink’s work [25] is then discussed in more detail.

**Chandy, Misra, and Sanders’ logics for UNITY programs:** The main trouble with Chandy and Misra’s logic [5] is the non-equivalence between the axiomatic and (informal in [5]) operational semantics of properties and the resulting confusion between inv and invariant properties. These semantics have been reconciled by Sanders [22], who weakened the properties by introducing strongest invariants in their axiomatic semantics. The proof system in [22] is complete but the compositionality of safety properties in [5] is lost because the strongest invariants of [22] are defined for components which are executed in isolation (no environment actions). This reveals a tradeoff between completeness and compositionality (see also the discussion on inv versus invariant properties in Section 4).

On the one hand, the unless properties of [5] can be retrieved by fixing $r = random$, which forces the strongest invariant to be True (even when programs do have initial
conditions on their shared variables). Properties with \( r = \text{random} \) do compose well but they are so strong that programs enjoy few of them; weaker properties cannot be expressed. On the other hand, the \textbf{unless} properties of [22] can be retrieved by fixing \( r = \text{skip} \) (or by considering all variables local). Properties with \( r = \text{skip} \) do not compose well but are much weaker and programs enjoy many of them. The tradeoff between completeness and compositionality is thus also a tradeoff between expressiveness and compositionality. This also appears in the treatment of progress properties: the composition rules of [5, 6, 22] are restricted to strong progress properties. Typically, the poorly composable property \( P \text{leadsto} Q \) (if \( P \) holds then \( Q \) holds eventually) has been replaced in [6] with a stronger composable progress property: if \( P \) holds, then \( Q \) holds eventually and continues to hold afterwards.

This paper has shown that there is no need for a tradeoff; in particular, the \textbf{invariant} properties compose well (Rule 50) and can be substituted for \textbf{True} in deriving other properties (Rule 28). The proof system includes compositional rules for safety (\textbf{invariant, guarantee}) and progress (\textbf{leadsto}) properties and is complete. The ability to cope with hiding is another improvement because of the implementation freedom gained by the presence of local variables: programs are specified with observable properties and these do not depend on the local variables.

\textit{Dijkstra's DUALITY processes}: DUALITY is a language introduced by Dijkstra [11] for the description and analysis of UNITY. Like UNITY programs, open systems can be mapped to DUALITY processes and their properties analysed using the operators of [11]. Unfortunately, there is also a tradeoff there. The mapping to DUALITY requires the initial conditions to be dropped (as in [6]), in which case \( \text{si}.O = \text{true} \) and the properties clearly have a different semantics. Alternatively, \( P \subseteq Q \) can be redefined as \( P.\sigma \Rightarrow Q.\sigma \) for all reachable states of \( O \), in which case parallel composition cannot be analysed because the order on predicates in a composite program is then different from the order on predicates in its components.

\textit{Udink's thesis}: This work on \textbf{specification refinement} and Udink's work on \textbf{program refinement} complement each other; both are concerned with achieving compositionality in a UNITY-like environment. Establishing the exact connection between these two approaches still requires further work but efforts to narrow the gap have been made in this paper, thus strengthening their similarities. The properties of [25] have no rely conditions but it has been shown that adding rely conditions amounts to considering open systems with an environment action different from \textbf{random}; the presence of an environment action is precisely the key difference between UNITY programs and the \textbf{ImpUNITY} programs of [25]. This suggests an interesting compositional approach to the development of concurrent programs. First, properties with rely condition can be refined (and in particular decomposed) using the techniques presented in this paper until they are specific enough; they can then be developed into \textbf{ImpUNITY} programs whose external action is precisely the rely condition used in their specification; finally, programs can be refined using the techniques of [25] which exploit the extra freedom given by the presence of local variables. An advantage of that combined approach
would be that the cooperation condition is verified at the specification level and thus still holds after properties have been developed into programs.

The logic proposed in this paper might be a better companion to the program refinement techniques of [25] than the specification logic of [25] because the latter has kept some of the deficiencies of the previous variants. In particular, there are no composition theorems for \textit{invariant} and \textit{leadsto} properties in [25] similar to Theorems 42 and 45. The absence of a composition theorem for \textit{leadsto} can probably be explained by the omission of \textit{e. O} in the semantics of \textit{ensures} properties in [25]. Instead, Udink has proposed a new composable progress property. This new property is however closer to \textit{ensures} than to \textit{leadsto} and has no known interpretation in terms of computations. This contrasts with the observation that the properties in this specification logic can all be given an equivalent operational semantics [8, 9].

10. Conclusion

Although clearly geared towards UNITY, this paper has tentatively presented a foundation for modular reasoning on state-based concurrent programs in general, with a simple temporal logic. Special attention has been given to a stepwise presentation of the composition rules. In essence, the composition rules follow from the relation between the strongest invariants in a composite system (Theorem 41), and from the conditions required to establish \((J_1, r_1, O_1) \Rightarrow (J_2, r_2, O_2)\) from \(O_1 \Rightarrow \neg O_2\) (Theorem 48).

References