Modular Formal Verification of Specifications of Concurrent Systems

Sara Gradara*, Antonella Santone*, Gigliola Vaglini*, Maria Luisa Villani*

* RCOST - Research Centre on Software Technology
University of Sannio, Benevento, Italy
{gradara,santone,villani}@unisannio.it

+ Dipartimento di Ingegneria della Informazione,
University of Pisa, Italy
g.vaglini@iet.unipi.it

Contact address:
Antonella Santone
RCOST - Research Centre on Software Technology
University of Sannio,
Palazzo ex Poste, Via Traiano 1 82100 Benevento, Italy

Abstract. The paper proposes a bottom-up approach to the verification of systems with modular structure: when modules are composed in specific ways, the complete software system verifies a composition of the properties each component does. The focus of the work is on the process of upgrading systems with new functionalities, where the validity of old requirements needs to be ensured, but also an understanding of the new properties the upgraded system would enjoy is useful. Each component is supposed to be specified by a CCS process, and the properties expressed by selective mu-calculus formulae.

Keywords: modular verification, model checking, CCS, logic.

1 Introduction

The use of automatic methods like model checking [10] in system verification is limited by the state explosion problem caused by the high number of concurrent components of the system. Nevertheless, techniques and methods that permit the modularization of the verification process of a large system can be employed. These techniques permit the verification of components in isolation (i.e. each component is independently verified, usually relying on a simplified model of the rest of the system (see, for example, Alur et al. [1], and Manna
et al. [26]) and allow inference of global properties of the whole system by composing the results in some way (see, for example, Li et al. [24]). From the point of view of the methodology, there are techniques based on the concept of refinement (essentially, they exploit decomposition or a top-down development of the verification) and techniques based on the concept of module composition/reuse (essentially, they use composition or a bottom-up development of the verification). Another important aspect of a compositional method is whether it is defined in terms of the syntactic structure of the language used in the formal description of the model, or, on the contrary, the method relies only on semantic concepts such as safety/liveness characterisations and closure.

The most important techniques for the modularization of the verification process can be grouped into four main categories (see also the work by Furia [18]), that can be listed from largely automatable to requiring a substantial human assistance. These techniques are partitioning and lazy composition, based on the concept of decomposition; interface abstraction, based on the concept of replacing a process \( p \) by a simpler one which is equivalent (examples are bisimulation and stuttering equivalence) to \( p \) from the point of view of the communications; the last category is based on the assume-guarantee techniques. The assume-guarantee reasoning allows the separate verification of each component, the environment is assumed to behave in a certain manner described by a logic formula, and a suitable inference rule permits to infer the global validity of the whole system. As general considerations, decomposing the system specification into parts, and identifying the sub-properties each part should verify, may not be an easy task; nevertheless, once both the system and the formula are expressed in terms of simpler components and corresponding sub-formulae, the whole verification process does not need to be repeated each time changes are made, but only those components directly impacted by the changes can be verified again.

Other approaches are based on given sufficient conditions for the target formula to be preserved (see, for example, Clarke et al. [11], and Santone [32]), when new components are added in parallel to incrementally design the system.

This paper proposes a bottom-up approach to verification, that is, component properties are separately verified, and global properties for the whole system are deduced, according to the interaction model underlying the components composition, i.e., whether the composition type is a parallel, or a sequence, or a choice. In fact, the work focuses on the process of upgrading systems with new functionalities, where the validity of old requirements needs to be ensured, but also an understanding of the new properties that the upgraded system would enjoy is useful. The approach is fully automatic. Here systems are described by the Calculus of Communicating Systems (CCS) [27] and properties are defined by logic formulae. Given two processes and the properties they verify, it is defined when those properties do not interfere with each other; for this purpose only the alphabet of the formulae and of the CCS processes is exploited. Thus the method is syntax-based: in particular, it relies on property specifications given through the selective mu-calculus logic (see Barbuti et al. [4]). The definition of
this logic exploits a concept similar to that of slices in the programming field (for a survey on program slicing see the paper by Tip [34]), where the slicing criterion is based on the given formula and permits an easy property-based construction of the abstract model. It is worth noting that the method only requires the specification of the point where the new process is added, and of the operator used, along with the properties to be composed, while the complete description of the two CCS parts is not used. Moreover, any type of property is in general allowed to hold for the sub-parts and the method enlightens the way in which the old formula is satisfied by the extended system.

CCS processes can represent the system specification in the design phase or they can be obtained from a Java program (see, for example, Chen [9], and Gradara et al. [20]), allowing in this case for the verification of a formal description of the system implementation by means of existing model checking environments, like the Concurrency Workbench of New Century (CWB-NC) [12]. In fact, modern software development is driven by key concepts such as: separation of concerns that leads to modular structures where components (or classes) are developed according to the features (or services) they have to provide; and extensibility, ensuring easy integration of new functions or new implementations of these functions. This implies the need for means of integration of the individual verification results not just limited to parallel compositions, as well as of reuse of the verification work whenever new features, or new implementations of old features, are added to the system.

Furthermore, the emerging field of web services engineering may suggest an interesting and viable application for a modular verification technique. Web services are distributed, independent, stateless components that can be accessed through the internet and communicate with the outside world by exchange of messages. New services can be constructed out of the existing ones, by specifying their interaction through some process description language such as WSBPEL [3], which is machine executable. Automatic translators from WSBPEL to process algebras like CCS have been developed (see, for example, Koshkina and van Breugel [23]) for verification purposes. In web service compositions there is a clear separation of the invoked services from the interaction logic, and the verification of the individual services is almost replaced by trust of their properties.

The paper is organised as follows: Section 2 briefly reviews the main concepts of CCS [27] and the selective mu-calculus [4], Section 3 describes the methodology that is illustrated through the example of Section 4. Finally, considerations and comparisons with some related work are given in Section 5.

2 Background

This section contains a brief overview both of the specification language that describes the system behavior and of the temporal logic used to define the system properties. In the first sub-section it is also defined well-termination of processes,
while some examples in the second sub-section illustrate the way in which the particular temporal logic used here allows a space reduction of the system model.

2.1 The Calculus of Communicating Systems (CCS)

CCS [27] is a specification language widely used for concurrent and distributed systems: CCS processes are described by means of the syntax

\[ p ::= \text{nil} \mid x \mid \alpha.p \mid p + p \mid p[p] \mid p \setminus L \mid p[f] \]

where \( \mathcal{A} = \{\tau, a, \bar{a}, b, \bar{b}, \ldots\} \) is a finite set of actions and \( \alpha \in \mathcal{A} \). The action \( \tau \in \mathcal{A} \) is called the internal action. The set of visible actions, \( \mathcal{V} \), ranged over by \( l \), is defined as \( \mathcal{A} - \{\tau\} \). Each action \( l \in \mathcal{V} \) has a complementary action \( l \). In processes of the form \( p \setminus L \), the set \( L \) is the set of actions such that \( L \subseteq \mathcal{V} \). The relabelling function \( f \) is a total function \( f : \mathcal{A} \rightarrow \mathcal{A} \) such that the constraint \( f(\tau) = \tau \) is satisfied. \( x \) is a constant name: each constant \( x \) is defined by a constant definition \( x \triangleq p \), where \( p \) is called the body of \( x \). A process is finite if it contains only finite summation and no constants (or recursions) (see [27]). The set of all processes is denoted by \( \mathcal{P} \). Given \( L \subseteq \mathcal{V} \), \( L \) denotes the set \( \{\alpha, \alpha \mid \alpha \in L\} \).

A labelled transition system, called \( S(p) \), gives the operational semantics of a process \( p \). \( S(p) \) is an automaton whose states correspond to processes (the initial state corresponds to \( p \)) and whose arcs are labelled by actions in \( \mathcal{A} \) and correspond to transitions from state to state. Such structural operational semantics are given by a relation \( \longrightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P} \). The relation \( \longrightarrow \) (\( \longrightarrow_e \) for short) is the least one defined by the rules in Table 1.

<table>
<thead>
<tr>
<th>Act</th>
<th>( \alpha.p \longrightarrow p )</th>
<th>Sum</th>
<th>( p \longrightarrow p' ) (and symmetric)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Con</td>
<td>( \frac{x \longrightarrow p'}{x \longrightarrow p'} )</td>
<td>Par</td>
<td>( p + q \overset{\alpha}{\longrightarrow} p' )</td>
</tr>
<tr>
<td></td>
<td>( \frac{x \overset{\alpha}{\longrightarrow}}{x \overset{\alpha}{\longrightarrow}} )</td>
<td></td>
<td>(and symmetric)</td>
</tr>
<tr>
<td>Com</td>
<td>( \frac{p \overset{\tau}{\longrightarrow} p', q \overset{\tau}{\longrightarrow} q'}{p[q] \overset{\tau}{\longrightarrow} p'[q']} )</td>
<td>Rel</td>
<td>( p[f] \overset{\tau(\alpha)}{\longrightarrow} p'[f] )</td>
</tr>
<tr>
<td></td>
<td>( p \overset{\alpha}{\longrightarrow} p' )</td>
<td></td>
<td>( p[L] \overset{\alpha}{\longrightarrow} p'[L] )</td>
</tr>
</tbody>
</table>

Table 1. Operational semantics of CCS.

The process \( \alpha.p \) can perform the action \( \alpha \) becoming the process \( p \) (rule Act). Sum states that \( p \) and \( q \) are alternative choices for the behaviour of \( p + q \).
**Par** shows how two parallel processes can behave autonomously, while **Com** defines how two processes communicate so performing the internal action \( \tau \). **Res** restricts the set of actions that can occur. **Rel** renames actions by means of the relabelling function \( f \). **Con** states that the constant \( x \) behaves as \( p \) if \( x \overset{\text{def}}{=} p \). No rule involves the process \( \text{nil} \) since it does not perform any action.

A CCS program is a pair \( (E, p) \), where \( p \), called *initial process*, belongs to \( P \) and \( E \) is a finite set of constant definitions.

Let \( p, q \) and \( r \) be CCS process, \( p[q/r] \) denotes the new CCS process obtained by substituting each sub-process \( r \) of \( p \) with \( q \).

The *sort* of a process \( p \) can be characterised as follows.

**Definition 1 (sort).** Let \( p \) be a CCS process and \( E \) be a set of constant definitions. \( L_E(p) \subseteq A \) is the set of actions obtained as the least solution of the following recursive definition:

\[
\begin{align*}
L_E(\text{nil}) &= \emptyset \\
L_E(\alpha.p) &= L_E(p) \cup \{\alpha\} \\
L_E(p|q) &= L_E(p) \cup L_E(q) \\
L_E(p\langle L) &= L_E(p) - \{\alpha, \tau \mid \alpha \in L\} \\
L_E(p[f]) &= \{f(\alpha) : \alpha \in L_E(p)\} \\
L_E(p + q) &= L_E(p) \cup L_E(q) \\
L_E(x) &= \begin{cases} \\
L_E(p) & \text{if } x \overset{\text{def}}{=} p \in E \\
\emptyset & \text{otherwise} \\
\end{cases}
\end{align*}
\]

When clear from the context, \( L(p) \) is used in place of \( L_E(p) \).

Later on, deadlocked processes are distinguished from well-terminating ones. For example, the process

\[(a.b.nil \langle \pi.nil)\rangle \setminus \{a, b\}\]

is a deadlocked process (since it cannot perform any action), while the process \( a.b.nil \) is well-terminating. Such distinction is needed to express sequentiality of processes. The action \( \delta \) is introduced to this aim, which is used to synchronise processes on termination as shown by the following definition.

**Definition 2 (Well-terminating process).** A CCS process is well-terminating if it performs an action \( \delta \) if and only if it terminates right after that.

In fact the typical use of \( \delta \) can be found in the definition of the operators \( \parallel \) and \( ";" \) (see Table 2). The two operators (defined as in Milner [27]) correspond to
the sequentialization and parallel execution of two well-terminating processes. The processes resulting from their applications are well-terminating too.

The constant DONE corresponds to a process whose task is to terminate without further moves.

The “;” operator will be suitably used to express the sequentiality of two processes p and q: p must terminate before q can start its execution. The process “p∥q” represents the parallel execution of p and q and terminates only if both processes terminate.

In the following, only CCS programs are considered in which the operator ∥ has been consistently substituted to |.

2.2 Model checking and selective µ-calculus

A model checker [10] accepts two inputs, a transition system and a temporal formula, and returns “true” if the system satisfies the formula, “false” otherwise; in the last case a counter-example may be produced, which is helpful to locate and correct errors.

The major problem in model checking is state explosion: complex systems are often described by transition systems with a prohibitive number of states. The primary cause of this problem is the parallel composition of interacting processes. When n processes of size (number of states) m are composed in parallel, the resulting process can be of size $m^n$.

The selective µ-calculus is a branching temporal logic to express behavioral properties of systems, introduced by Barbuti et al. [4]. It is equi-expressive to µ-calculus [33], but different in the definition of the modal operators. Given a set $\mathcal{A}$ of actions and a set Var of variables, selective µ-calculus formulae are defined as follows:

$$\varphi ::= \top | \bot | Z \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid [K]_R \varphi \mid (K)_R \varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

where $Z \in$ Var and $K, R \subseteq \mathcal{A}$. The operators $\mu Z.\varphi$ and $\nu Z.\varphi$ are fixed point operators: $\mu Z.\varphi$ is the least fixed point of the recursive equation $Z = \varphi$, while

\[
p \parallel q = (p[\delta_1/\delta] \mid q[\delta_2/\delta] \mid (\delta_1, \delta_2, DONE + \delta_2, DONE))\}\{\delta_1, \delta_2\}
\]

\[
p ; q = (p[\delta_3/\delta] \mid \delta_3, q)\}\{\delta_3\}
\]

\[
DONE = \delta . nil
\]

Table 2. Operators for well-terminating processes.
\( \nu Z, \varphi \) is the greatest one. In the formula \( \mu Z, \varphi \) \( (\nu Z, \varphi) \mu Z \) \( (\nu Z) \) binds the occurrences of \( Z \) in \( \varphi \). A variable that is not bounded by any fixed point operators is called free. A formula without free variables is called closed. From now on only closed formulae are considered.

The state \( s \) of a transition system satisfies the selective formula \( \varphi \), written \( s \models \varphi \), as follows:

- \( s \) always satisfies \( \mathbf{t} \mathbf{t} \) and never \( \mathbf{f} \mathbf{f} \); moreover, \( s \) satisfies \( \varphi_1 \lor \varphi_2 \) (\( \varphi_1 \land \varphi_2 \)) if it satisfies \( \varphi_1 \) or (and) \( \varphi_2 \).
- \( s \) satisfies:
  - \( [K]_R \varphi \) if, after every sequence of actions not belonging to \( R \cup K \), followed by an action in \( K \), evolves to a state obeying \( \varphi \).
  - \( (K)_R \varphi \) if, after some sequence of actions not belonging to \( R \cup K \), followed by an action in \( K \), evolves to a state obeying \( \varphi \).

A transition system satisfies \( \varphi \) if and only if \( s \models \varphi \), where \( s \) is its initial state. A CCS process \( p \) satisfies \( \varphi \) if \( \mathcal{S}(p) \) satisfies \( \varphi \). The precise definition of the satisfaction of the closed formula \( \varphi \) by the process \( p \) is given in Table 3: the transition relation \( \Rightarrow_1 \), parametric with respect to \( I \subseteq A \), is defined as follows and ignores all non-interesting actions (i.e. those in \( A - I \)).

**Definition 3 \( (\Rightarrow_1 \) relation).** Let \( p \) be a CCS process and \( I \subseteq A \), the relation \( \Rightarrow_1 \) is such that, for each \( \alpha \in I \), \( p \xRightarrow[\gamma]{\alpha} q \) iff \( p \xRightarrow{} q^\gamma \), for some \( \gamma \in (A - I)^* \).

Note that \( \Rightarrow_A = \xrightarrow{} \).

\[ \begin{align*}
p \neq \mathbf{f} \mathbf{f} & \quad p \models \mathbf{t} \mathbf{t} \\
p \models \varphi \land \psi & \iff p \models \varphi \text{ and } p \models \psi \\
p \models \varphi \lor \psi & \iff p \models \varphi \text{ or } p \models \psi \\
p \models [K]_R \varphi & \iff \forall p', \forall \alpha \in K, p \xRightarrow[\alpha]{\alpha} K \cup R \implies p' \models \varphi \\
p \models (K)_R \varphi & \iff \exists p', \exists \alpha \in K, p \xRightarrow[\alpha]{\alpha} K \cup R \text{ and } p' \models \varphi \\
p \models \nu Z, \varphi & \iff p \models \nu Z^n, \varphi \text{ for all } n \\
p \models \mu Z, \varphi & \iff p \models \mu Z^n, \varphi \text{ for some } n
\end{align*} \]

where, for each \( n \), \( \nu Z^n, \varphi \) and \( \mu Z^n, \varphi \) are defined as:

\[ \begin{align*}
\nu Z^0, \varphi = \mathbf{t} \mathbf{t} & \quad \mu Z^0, \varphi = \mathbf{f} \mathbf{f} \\
\nu Z^{n+1}, \varphi = \varphi[\nu Z^n, \varphi / Z] & \quad \mu Z^{n+1}, \varphi = \varphi[\mu Z^n, \varphi / Z]
\end{align*} \]

and \( \varphi[\psi / Z] \) indicates the substitution of \( \psi \) for each free occurrence of \( Z \) in \( \varphi \).

<table>
<thead>
<tr>
<th>Table 3. Satisfaction of a closed formula by a process.</th>
</tr>
</thead>
</table>

1. \( [K]_R \varphi \) is equivalent, using CTL logic [15], to \( A[!\{(R \text{ or } K) \cup W \text{ (K and } \varphi)\}] \).
2. If \( \gamma = \alpha_1 \ldots \alpha_n \), \( n \geq 1 \), \( p \xrightarrow[\gamma]{\alpha_1} q \) means \( p \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} q \).
Example 1. Some examples of selective mu-calculus formulae are given to explain the use of the selective operators.

\[ \varphi_1 = [b]_a \mathsf{ff} \text{: "b cannot be performed if a has not been performed before".} \]

\[ \varphi_2 = \langle b \rangle \mathsf{tt} \text{: "it is possible to perform } b \text{ without any restrictions on actions preceding } b". \]

\[ \varphi_3 = \nu Z. [a]_0(Z \land [a]_c \mathsf{ff}) \text{: "it always holds that, after } a \text{ has occurred, no successive } a \text{ can occur if } c \text{ has not occurred before".} \]

Consider the processes:

\[
\begin{align*}
x & \overset{\text{def}}{=} b.c.x + a.b.c.x \\
y & \overset{\text{def}}{=} a.b.c.y \\
z & \overset{\text{def}}{=} b.z + a.b.z
\end{align*}
\]

The transition systems for \( x, y \) and \( z \) are in Figure 1. It holds that:

\[
\begin{align*}
x & \not\models \varphi_1 \models \varphi_2 \models \varphi_3 \\
y & \models \varphi_1 \models \varphi_2 \models \varphi_3 \\
z & \not\models \varphi_1 \models \varphi_2 \not\models \varphi_3
\end{align*}
\]

Fig. 1. Three transition systems.

The main point is that, in contrast to mu-calculus, the selective calculus allows us to immediately point out, from each formula, the transitions and states of the transition system that do not alter the truth value of the formula itself. More precisely, the result of the checking only depends on the actions explicitly mentioned in the modal operators used in the formula (namely, occurring actions).

In Barbuti et al. [4], \( \rho \)-equivalence is defined to formally characterize the notion of “the same behavior with respect to a set \( \rho \) of actions”: two transition systems are \( \rho \)-equivalent if and only if satisfy the same set of formulae with occurring actions in \( \rho \).
Example 2. Recall the formulae of Example 1 and the transition systems in Figure 1. $S(x)$ is $\{a, b\}$-equivalent to $S(z)$, on the contrary $S(y)$ is not $\{a, b\}$-equivalent to $S(z)$. Thus $S(x)$ and $S(z)$ give the same result for the checking of the formulae with occurring actions in $\{a, b\}$; in particular, they satisfy $\varphi_2$, while they do not satisfy $\varphi_1$. Note that the set of occurring actions of $\varphi_1$ is $\{a, b\}$, while that of $\varphi_2$ is $\{b\}$ which is a subset of $\{a, b\}$.

As shown in Barbuti et al. [4], mu-calculus recursive formulae can be easily translated into selective formulae (and vice versa): the aim is to reduce the number of actions that appear in the mu-calculus formula. Exploiting the selective mu-calculus, improvements in model checking a process $p$ can be obtained by eliminating from $S(p)$ "most" of the actions that are not in $\rho$. Obviously, the reduction degree mainly depends on the size of $\rho$ compared to the size of $A$. The reduction of the transition system of a process $p$ can be obtained through a syntactic reduction of $p$. In Barbuti et al. [6], a CCS process $p$ is transformed into another process $q$ on which the selective formula $\varphi$ can be equivalently checked\(^3\). The method is based on a set of transformation rules that modify the description of $p$ by eliminating most of the actions not occurring in $\varphi$. Figure 2 shows the transition systems corresponding to the processes

\[
x' \overset{\text{def}}{=} b.x' + a.b.x' \quad y' \overset{\text{def}}{=} b.y' \quad z' \overset{\text{def}}{=} b.z' + a.z'
\]

obtained from $x, y, z$ through the reductions driven by $\varphi_1, \varphi_2, \varphi_3$, respectively.

The set of occurring actions of $\varphi_1$ is $\{a, b\}$, while that of $\varphi_2$ is $\{b\}$ and that of $\varphi_3$ is $\{a, c\}$. Note that the action $b$ in $z'$ is maintained to preserve the two choices of the $+$ operator.

\[\text{Fig. 2. Three reduced transition systems.}\]

In a modular approach to system verification, the separate verification of the properties of the transition systems corresponding to the different sub-modules

\(^3\) A similar method has been defined for LOTOS [7] processes in Barbuti et al. [5].
can achieve a better result in terms of memory space and time than the verifica-
tion of the transition system of the whole CCS process. It is worth noting that, on the other hand, a further improvement in terms of memory space can be achieved by building the transition systems of each module in a reduced form with respect to $\rho$-equivalence, taking into account only the actions in $\rho$.

3 Modular Verification of abstract CCS processes

The proposed methodology for the modular verification of CCS processes can be applied in the design phase of a concurrent system; otherwise, when managing programs, it can be applied after having translated a Java program [9, 20], for example, into a CCS specification. The methodology is also useful when system or program functionalities must be upgraded or extended; another possible case of use occurs when the mobility of the code can cause the reconnection of processes previously disconnected. In all these cases the system should not be completely redesigned and verified each time a modification occurs, but the result of the previous verification must be maintained and only the new part separately checked. This modular verification methodology can be integrated with the reduction methodology driven by the selective formulae; in fact, the verification of the previously existing system and of the new module can be performed exploiting their reduced transition systems that maintain only the occurring actions of each formula. So, through the modular and abstract verification technique, a memory space saving, beside the time saving, can be obtained.

3.1 Composition of processes

This section describes ways to add the functionalities supplied by a process $q$ to a process $p$, through a set of operators whose semantics are defined in terms of CCS operators. The aim is to separately verify the properties of $p$ and $q$ and deduce the properties verified by their composition. A particular event $a \in \mathcal{L}(p)$ is used as a marker for the activation point of the process $q$ chosen to augment the functionalities of $p$.

The possible compositions of the processes $p$ and $q$ are defined below. Suppose that $p$ and $q$ are constant processes, called $x$ and $y$ respectively, according to the CCS syntax in sub-section 2.1. The extension to the other cases is obvious. From now on, $y$ represents the process to be integrated within the process $x$. Roughly speaking, suppose $y$ and $x$ belonging to distinct sets of constant definitions, $\mathcal{E}_y$ and $\mathcal{E}_x$, respectively; a new constant $\tilde{x}$, built from one of the alternatives below, is substituted to $x$ in $\mathcal{E}_x$; moreover $\tilde{x}$ substitutes each occurrence of the name $x$ in the body of $x$, and of all the constants in $\mathcal{E}_x$. Finally $y$ and $\mathcal{E}_y$ will integrate $\mathcal{E}_x$ building a new environment. Three composition operators have been defined:

- $x \oplus_a y$: in each process $a.r$ occurring in the body of $x$, the choice $r+y$ will replace $r$;
- $x|_a^L y$, where $L \subseteq V$: in each process $a.r$ occurring in the body of $x$, the parallel composition $(r\parallel y)|L$, will replace $r$; when $L = \emptyset$, $x|_a^L y$ is used instead of $x|_a^L y$;
- $x.a y$: in each process $a.r$ occurring in the body of $x$, the sequential composition $y;p$ will replace $r$.

The formal definition of these operators is given in Definition 4.

**Definition 4 (Process composition operators).** Let $P_1 = \langle E_1, z_1 \rangle$, $P_2 = \langle E_2, z_2 \rangle$ be two CCS programs. Suppose that if $x \overset{\text{def}}{=} p_1 \in E_1$ and $x \overset{\text{def}}{=} p_2 \in E_2$, then $p_1 = p_2$, and consider $y \overset{\text{def}}{=} p_0 \in E_2$ and $x \overset{\text{def}}{=} p_0 \in E_1$.

Let $\{x_1, \ldots x_k\} = \{x_1, x_2, \ldots, x_k\} \overset{\text{def}}{=} p_2, p_1 \in E_1 \cup E_2$ and $x$ occurs in $p_2$. Then:

1. $x \oplus_a y$

   from this kind of composition the new program $P = \langle \text{new}E, \tilde{z}_1 \rangle$ can be obtained, where $\text{new}E = E_1 \cup E_2 - \{x\} - \{x_1, \ldots x_k\} \cup \{\hat{x}\} \cup \{\hat{x}_1, \ldots \hat{x}_k\}$ with:
   
   \[ \tilde{x} \overset{\text{def}}{=} p_x[a.(y + p)/a.p], \]
   
   $\tilde{x}_i \overset{\text{def}}{=} p_x[\hat{x}/x]$ for each $i \in [1..k]$;

2. $x|_a^L y$

   from this kind of composition the new program $P = \langle \text{new}E, \tilde{z}_1 \rangle$ can be obtained, where $\text{new}E = E_1 \cup E_2 - \{x\} - \{x_1, \ldots x_k\} \cup \{\hat{x}\} \cup \{\hat{x}_1, \ldots \hat{x}_k\}$ with:
   
   $\tilde{x} \overset{\text{def}}{=} p_x[a.(y\parallel p)|L/a.p]], \]
   
   $\tilde{x}_i \overset{\text{def}}{=} p_x[\hat{x}/x]$ for each $i \in [1..k]$;

3. $x.a y$

   from this kind of composition the new program $P = \langle \text{new}E, \tilde{z}_1 \rangle$ can be obtained, where $\text{new}E = E_1 \cup E_2 - \{x\} - \{x_1, \ldots x_k\} \cup \{\hat{x}\} \cup \{\hat{x}_1, \ldots \hat{x}_k\}$ with:
   
   $\tilde{x} \overset{\text{def}}{=} p_x[a.(y.p)/a.p], \]
   
   $\tilde{x}_i \overset{\text{def}}{=} p_x[\hat{x}/x]$ for each $i \in [1..k]$.

Some interesting uses of these process composition operators are pointed out below.

---

4 $\tilde{z}_1 = z_1$ if $z_1 \not\in \{x_1, \ldots x_k, x\}$

5 Given a set $\mathcal{E}$ of constant definitions, for the sake of simplicity, the set $\mathcal{E} - \{x\} \cup \{y\}$ denotes the new set of constant definitions obtained from $\mathcal{E}$ by eliminating the definition for $x$ and introducing that for $y$, i.e., $\mathcal{E} - \{x \overset{\text{def}}{=} p_x\} \cup \{y \overset{\text{def}}{=} p_y\}$. 

11
1. The operator $\oplus_a$ is useful to represent the addition of a new sub-choice to a menu, after the marker event $a$; obviously this new choice does not interfere with the other choices of the menu.
2. The operator $|_a$ is useful to represent the introduction of a new service/user in a service system, or the reconnection of a mobile process to the system after a disconnection, after the marker event $a$. This new process can interfere with the other processes in the system.
3. The operator $\cdot_a$ is useful to represent a procedure call after the marker event $a$. The call produces the execution of the process $q$ after $a$, but before the events following $a$ in the system.

The following example shows the result of some process compositions.

**Example 3.** Following Definition 4, the results below are obtained.

1. Suppose $P_1 = \langle \{ \text{def } x = a.b.b.x \}, x \rangle$ and $P_2 = \langle \{ \text{def } y = b.b.d.y \}, y \rangle$.

   From $x|_a y$, the new program

   \[ P = \langle \{ \hat{x} = a.(b.b.\hat{y}) \}, \hat{y} = b.b.d.y, \hat{x} \rangle \]

   is obtained.

2. Suppose $P_1 = \langle \{ \text{def } x = a.b.DONE \}, x \rangle$ and $P_2 = \langle \{ \text{def } y = c.DONE \}, y \rangle$.

   From $x|_a y$, the new program

   \[ P = \langle \{ \hat{x} = a.(b.DONE \hat{y}) \}, \hat{y} = c.DONE, \hat{x} \rangle \]

   is obtained, and, from $x\cdot_a y$, the new program

   \[ P = \langle \{ \hat{x} = a.(y.b.DONE) \hat{y} = c.DONE \}, \hat{x} \rangle \]

3. Suppose $P_1 = \langle \{ \text{def } x = a.b.DONE \}, x \rangle$ and $P_2 = \langle \{ \text{def } y = (d.b.DONE) \hat{y} = (b) \}, y \rangle$.

   From $x\cdot_a y$, the new program

   \[ P = \langle \{ \hat{x} = a.y; b.DONE \hat{y} = (d.b.DONE) \hat{y} = (b) \}, \hat{x} \rangle \]

   is obtained. It is worth noting that $y$ is not well-terminating.

### 3.2 Composition of formulae

Consider two CCS processes $p$ and $q$, such that $p \models \varphi$ and $q \models \psi$. The composition of $p$ and $q$, through the operators defined in the previous section, satisfies a formula obtained by composing $\varphi$ and $\psi$ through the operators of the selective mu-calculus. To this end, the alphabet of $p$ and $q$, before any possible syntactic reduction, must satisfy a further condition expressed by Definition 6 on the sets of actions of Definition 5.
Definition 5 (box(φ), sub(φ)). Let φ be a selective mu-calculus formula and $L \subseteq A$ an alphabet. Then:

$$\text{box}(\varphi) = \{ \alpha \mid [K]_R \text{ occurs in } \varphi, \alpha \in K \}$$
$$\text{sub}(\varphi) = \{ \alpha \mid \langle K \rangle_R \text{ occurs in } \varphi, \alpha \in R \}.$$ 

Let φ be a selective mu-calculus formula, the set box(φ) contains all actions occurring in the set K of some modal operators $[K]_R$ in φ; while the set sub(φ) contains all actions occurring in the set R of some modal operators $\langle K \rangle_R$ in φ.

Given a CCS process $p$ and a marker event $a$, the set $L^a(p)$, which is a subset of $L(p)$, is defined as:

$$L^a(p) = \{ \beta \mid \exists q \text{ s.t. } p \xrightarrow{\gamma \alpha \gamma'} q \text{ with } \gamma \in (A - \{a\})^*, \gamma' \in A^* \}.$$ 

Roughly speaking, $L^a(p)$ contains all the actions that could be performed by $p$ after the marker $a$. Note that a syntactic definition of $L^a(p)$ is possible, but the result would be less accurate due to the difficulty of considering all possible thread interleavings.

Definition 6 (Conditions $B$ and $D$). Consider the processes $p$ and $q$ such that $p \models \varphi$ and $q \models \psi$. The following conditions on $\varphi$ and $\psi$ can be defined.

$$B = (\text{box}(\varphi) \cap L(q)) = \emptyset \land (\text{sub}(\psi) \cap L(p)) = \emptyset$$
$$D = \text{sub}(\varphi) \cap L(q) = \emptyset.$$ 

Definition 6 is motivated by the need to exclude some dangerous scenarios. Namely, condition $B$ ensures that process $q$ does not introduce sequences of actions forbidden by the formula $\varphi$, e.g., if $\varphi$ is a safety property. Consider the property $\varphi = [b]_c \mathsf{ff}$ and suppose that $p$ satisfies $\varphi$. Then requiring that $q$ is not able to perform $b$ is enough to avoid the computations where $b$ is not preceded by $c$. Instead, the fact that $q$ be able to perform $c$ is not relevant for the validity of $\varphi$. Similarly, if $\psi$ is composed with $\varphi$, i.e., the marker action $a$ belongs to a box or diamond operator of $\varphi$, the part of $p$ following the action $a$ does not have to impact the validity of the box modal operators in $\psi$. For example, suppose that $q$ satisfies $\psi = [d]_b \mathsf{ff}$. If $p$ was able to perform $d$ at some state after performing $a$, $\psi$ could not be valid on the part of the composition of $p$ with $q$ that follows $a$. Instead, the ability of $p$ to perform $d$ before the first occurrence of $a$ is not relevant for the validity of $\psi$. Furthermore, condition $D$ ensures that order relationships of actions required by $\varphi$ are preserved in the composition of $p$ with $q$. Consider the property $\varphi = [b]_c \mathsf{tt}$ and suppose that $p$ satisfies $\varphi$. If the process $q$ is able to perform $c$ and it is composed with $p$ by the sequence operator, then the action $b$ performed by $p$ should not follow $c$ in the process composition. The following example is intended to further clarify Definition 6.

$^6$ If $a$ does not syntactically occur in $\varphi$, then condition $B$ can be furtherly reduced as $B = \text{box}(\varphi) \cap L(q) = \emptyset$. 

13
Example 4. Consider again the programs of Example 3.

1. For $P_1 = \langle \{ x \overset{\text{def}}{=} a.b.b.x \}, x \rangle$ and $P_2 = \langle \{ y \overset{\text{def}}{=} b.b.d.y \}, y \rangle$. From $x \mid_{a(b)} y$, the new program

$$P = \langle \{ \tilde{x} \overset{\text{def}}{=} a.(b.b.\tilde{x}) \{ y \}, y \overset{\text{def}}{=} b.b.d.y \}, \tilde{x} \rangle$$

can be obtained. Since $x \mid_{a} = \varphi = \langle a \rangle_{\emptyset} \top \top$; and $y \mid_{b} = \psi = \nu Z. \langle (a, d, \tau)_{\emptyset} \top \top \wedge [a, d, \tau]_{\emptyset} Z \rangle$

and also $L(\tilde{x}) = \{ a, d, \tau \}$, the formula $\psi$ expresses the deadlock-freeness of $\tilde{x}$.

It holds that $L(x) = \{ a, b \}$, $L(y) = \{ b, d \}$ and $L^a(x) = \{ b \}$; thus $\text{box}(\varphi) \cap L(y) = \emptyset$, and $\text{box}(\psi) \cap L^a(x) = \emptyset$. Condition $B$ is fulfilled.

2. Now, consider $P_1 = \langle \{ x \overset{\text{def}}{=} a.b.DONE \}, x \rangle$ and $P_2 = \langle \{ y \overset{\text{def}}{=} c.c.DONE \}, y \rangle$.

It is true that $x \mid_{a} = \varphi = \langle a \rangle_{\emptyset} \top \top (b)_{\{ c \}} \top \top$; and $y \mid_{c} = \psi = \langle c \rangle_{\emptyset} \top \top$.

Moreover, $L(x) = \{ a, b \}$, $L^a(x) = \{ b \}$, $L(y) = \{ c \}$, $\text{box}(\varphi) \cap L(y) = \emptyset$, $\text{box}(\psi) \cap L^a(x) = \emptyset$, and $\text{sub}(\varphi) \cap L(y) = \{ c \}$. Thus, condition $B$ is obviously fulfilled, while $D$ is not.

3. Finally, for $P_1 = \langle \{ x \overset{\text{def}}{=} a.b.DONE \}, x \rangle$ and $P_2 = \langle \{ y \overset{\text{def}}{=} d.b.DONE \{ b \} \}, y \rangle$,

it is true that $x \mid_{a} = \varphi = \langle a \rangle_{\emptyset} \top \top (b)_{\emptyset} \top \top$; and $y \mid_{b} = \psi = \langle d \rangle_{\emptyset} \top \top$.

Thus, conditions $B$ and $D$ are both obviously fulfilled by $\varphi$ and $\psi$.

Relying on Definition 6, it is possible to separately verify two processes for their properties and maintain them after the composition. For this purpose and for the sake of clarity, suppose that the marker for the activation $a \in L(p)$ does not occur relabelled in $p$. This is not a limitation; to include also this case, it is sufficient to replace $a$ with $f^{-1}(a)$ in the formula composition definition given below. Note that, to apply the method described, the complete description of the process to be integrated is not required; it is sufficient to know only its alphabet.

The following definition denotes a new operator in the selective mu-calculus, which corresponds to the sequential composition of formulae.

\[ f^{-1}(S) = \{ \alpha \mid f(\alpha) \in S \}. \]
Definition 7 (The formula sequence operator). Let \( \varphi \) and \( \psi \) be two selective formulae. The formula sequence operator \( \langle \cdot \rangle \) is defined as: \( \varphi ; \psi \triangleq \varphi \langle \psi / \tt \rangle \), with the effect of replacing each occurrence of \( \tt \) in \( \varphi \) with the formula \( \psi \).

Definition 8 (Formulae composition). Let \( \varphi \) and \( \psi \) be two selective formulae. Their composition is described as follows:

1. \( \varphi \circ_\alpha \psi \) is obtained by substituting each sub-formula \( [a]_R \varphi \) \(^8\) (resp. \( \langle a \rangle_R \varphi \) ) in \( \varphi \) with \( [a]_R(\psi \lor \varphi') \) (resp. \( \langle a \rangle_R(\psi \lor \varphi') \) ),
2. \( \varphi \mid_\psi \) is obtained by substituting each sub-formula \( [a]_R \varphi' \) (resp. \( \langle a \rangle_R \varphi' \) ) in \( \varphi \) with \( [a]_R(\psi \land \varphi') \) (resp. \( \langle a \rangle_R(\psi \land \varphi') \) ),
3. \( \varphi \times \psi \) is obtained by substituting each sub-formula \( [a]_R \varphi' \) (resp. \( \langle a \rangle_R \varphi' \) ) in \( \varphi \) with \( [a]_R(\psi; \varphi') \) (resp. \( \langle a \rangle_R(\psi; \varphi') \) )

Now some other notions follow that are useful to present the main result of the paper. A process \( p' \) such that \( p \xrightarrow{a_0} p' \) is an \( a \)-derivative of \( p \). The set of all \( a \)-derivatives of \( p \) is denoted by \( D^a(p) \). The notion of \( \langle \omega, \rho \rangle \)-simulation is introduced in a paper by Santone [32].

Definition 9 (\( \langle \omega, \rho \rangle \)-simulation). Let \( p \) and \( q \) be two CCS processes and \( \omega, \rho \subseteq \Psi \).

- A \( \langle \omega, \rho \rangle \)-simulation, \( S \), is a relation on \( P \times P \) such that \( p \succeq \omega, \rho \) implies:

\[
\frac{p \xrightarrow{a} p'}{q \xrightarrow{a} q'}
\]

for some \( q' \) with \( p' \succeq \omega, \rho \).

- The process \( q \) \( \langle \omega, \rho \rangle \)-simulates \( p \) (write \( p \preceq q \)) if and only if there exists a \( \langle \omega, \rho \rangle \)-simulation \( S \) containing the pair \( (p, q) \).

Suppose that \( \omega \) is a set of communication actions, \( q \langle \omega, \rho \rangle \)-simulates \( p \) means that, if \( p \) becomes the process \( p' \), after performing a sequence of actions which are not communication actions, followed by an action \( \alpha \) in \( \omega \), then \( q \) performs \( \tau \), without previously executing any action in \( \omega \cup \rho \), and becomes a process which \( \langle \omega, \rho \rangle \)-simulates \( p' \). Roughly speaking, if \( p \preceq q \) then the process \( q \) does not block the execution of \( p \); moreover \( q \) does not execute actions in \( \rho \) before \( p \) does.

Note that \( q \) can have more computations than \( p \).

Let \( \varphi \) be a selective formula. The set \( \text{alph}(\varphi) \) denotes the alphabet of the formula \( \varphi \), i.e. \( \text{alph}(\varphi) = \{ \alpha \mid [K]_R \psi \text{ or } \langle K \rangle_R \psi \text{ occurs in } \varphi \text{ and } \alpha \in K \cup R \} \).

The main result of the paper is presented in the following theorem.

\(^8\) \( [K]_R \varphi \) and \( \langle K \rangle_R \varphi \) are shorthand for

\[
[K]_R \varphi = \bigwedge_{\alpha \in K} [\alpha]_R \varphi \quad \langle K \rangle_R \varphi = \bigvee_{\alpha \in K} \langle \alpha \rangle_R \varphi.
\]

Thus, given an action \( \alpha \in K \), \( [K]_R \varphi = [\alpha]_R \varphi \land [K - \alpha]_R \varphi \); and \( \langle K \rangle_R \varphi = [\alpha]_R \varphi \lor \langle K - \alpha \rangle_R \varphi \).
Theorem 1. Let $x$ be a CCS process satisfying the property $\varphi$ and $y$ a CCS process satisfying $\psi$. Moreover, suppose that $x$ and $y$ satisfy condition $B$. It holds that:

1. $x \oplus_a y \models \varphi \oplus_a \psi$.
2. $x \rightarrow^L_a y \models \varphi \rightarrow^a \psi$, if $(\text{alph}(\varphi) \cup \text{alph}(\psi)) \cap L = \emptyset$, $r \subseteq_{\text{sub}(\varphi)}^L y$, $y \subseteq_{\text{sub}(\psi)}^L r$ for each $r \in \mathcal{D}^a(x)$.
3. If $y$ is a well-terminating finite process and $x$ and $y$ satisfy condition $D$ too, then $x \rightarrow_a y \models \varphi \rightarrow_a \psi$.

Proof Sketch: By induction on the structure of the formula $\varphi$. Consider first the case $x \oplus_a y \models \varphi \oplus_a \psi$.

Base step. tt, ff: straightforward.

Inductive step.

$\varphi = (\alpha)_R \varphi'$: Since $x \models (\alpha)_R \varphi'$, this means that there exists a subprocess of $x$ with the form $\alpha.r$ such that $r \models \varphi'$. By definition of $x \oplus_a y$ there must exist a subprocess in $\bar{x}$ with the form $(\alpha.r)[a.(y + p')/a.p']$, thus from the inductive hypothesis it holds that $r \oplus_a y \models \varphi' \oplus_a \psi$, thus $\bar{x} \models \varphi \oplus_a \psi$.

$\varphi = [\alpha]_R \varphi'$: Since $x \models [\alpha]_R \varphi'$, this means that each process $r$, with $\alpha.r$ subprocess of $x$, satisfies $\varphi'$. By definition of $x \oplus_a y$, instead of each process $\alpha.r$, there must exist a subprocess in $\bar{x}$ with the form $(\alpha.r)[a.(y + p')/a.p']$. From the inductive hypothesis, it holds that $r \oplus_a y \models \varphi' \oplus_a \psi$. Since condition $B$ holds (i.e. box($\varphi$) \cap $L(y) = \emptyset$), the action $\alpha$ does not occur in the sort of $y$. Moreover, if $[\beta]_R \xi$ occurs in $\psi$, since condition $B$ holds (i.e. box($\psi$) \cap $L^d(x) = \emptyset$), $p'$ does not does not perform the action $\beta$. Thus, to prove the theorem ($\bar{x} \models \varphi \oplus_a \psi$) it is sufficient that $\psi$ holds on $y$ and that $\varphi'$ holds on each process $r$: both are true by inductive hypothesis.

$\varphi = \varphi' \lor \varphi''$, $\varphi = \varphi' \land \varphi''$: straightforward by inductive hypothesis.

$\varphi = \mu\alpha.\varphi'$, $\varphi = \nu\alpha.\varphi'$: the thesis follows since the truth value of such formulae corresponds to the $\lor/\land$ of an enumerable set of finite non-recursive formulae. In fact, the processes considered here are finite and finitely branching [27].

The case $x \rightarrow^L_a y \models \varphi \rightarrow^a \psi$ can be proved similarly, using the proofs made in Santone [32]. Consider now the case $x \rightarrow_a y \models \varphi \rightarrow_a \psi$. Only the case of the diamond operator is proved, all the other cases can be proved similarly.
\[ \varphi = (\alpha)_R \psi' : \text{Since } x \models (\alpha)_R \psi', \text{ this means that there exists a subprocess of } x \text{ with the form } \alpha.x \text{ such that } r \models \psi'. \text{ By definition of } x.a_y \text{ there must exist a subprocess in } \tilde{x} \text{ with the form } (\alpha.r)[a.(y;p')/a.p']. \text{ The } \varphi.a\psi \text{ is obtained by substituting each subformula } (\alpha)_R \xi \text{ in } \varphi \text{ with } (\alpha)_R(\psi; \xi). \text{ By inductive hypothesis, } y \models \psi \text{ and, since } y \text{ is finite and well-terminating, the actions in } r \text{ will be performed, so } \varphi' \text{ still holds, moreover, since condition } D \text{ holds, the actions in } R \text{ will never be performed by } y. \]

Let us explain intuitively the case \( x |_a^L y \models \varphi |_a \psi \). When considering a non empty set \( L \), no action in \( T \) must occur in both formulae, since the new composed process could not perform that action. Moreover, the \( (\omega, \rho) \)-simulation must be checked among \( y \) and all the \( a \)-derivatives of \( x \), since the process \( y \) is integrated within the process \( x \) after the marker point \( a \). The following examples better explain the results of the above theorem.

**Example 5.** Consider again the processes \( x \) and \( y \) of Examples 3 and 4.

1. Given \( P_1 = (\{x \text{ def} = a.b.b.x\}, x) \) and \( P_2 = (\{y \text{ def} = b.b.d.y\}, y) \),

from \( x |_a^{b_L} y \), the new program

\[ P = (\{\tilde{x} \text{ def} = a.(b.b.x || y)\}^{b_L}, y \text{ def} = b.b.d.y, \tilde{x}) \]

can be obtained. The check of \( \tilde{x} \) for the verification of the composition of the formulae \( \varphi \) and \( \psi \), that is,

\[ \varphi |_a \psi = (\alpha)_a(\tau \land \nu \langle b \rangle).((\alpha, d, \tau) \emptyset \tau \land [a, d, \tau] \emptyset Z) \]

gives result true. In fact, Example 4 shows that condition \( B \) is fulfilled, \( b.b.x \leq^{(b)} y \) and \( y \leq^{(b)} b.b.x \). Note that \( b.b.x \) is the only \( a \)-derivative of \( x \).

2. Given \( P_1 = (\{x \text{ def} = a.b.DONE\}, x) \) and \( P_2 = (\{y \text{ def} = c.c.DONE\}, y) \),

from \( x |_a y \), the new program

\[ P' = (\{\tilde{x} \text{ def} = a.(b.DONE || y), y \text{ def} = c.c.DONE\}, \tilde{x}') \]

is obtained and, from \( x.a_y \), the new program

\[ P'' = (\{\tilde{x}'' \text{ def} = a.(y; b.DONE), y \text{ def} = c.c.DONE\}, \tilde{x}'') \]

follows. The check of \( \tilde{x}' \) of \( P' \) and \( \tilde{x}'' \) of \( P'' \) for the verification of the composition of formulae \( \varphi \) and \( \psi \), i.e.,

\[ \varphi |_a \psi = (\alpha)_a(\langle b \rangle, \langle c \rangle \emptyset \tau \land \langle c \rangle \emptyset .) \]
\[ \varphi \cdot \psi = \langle a \rangle_0 \langle c \rangle_0 \langle b \rangle_0 \langle c \rangle_0 \top, \]

gives as result that the process \( a.(b.DONE || y) \) satisfies \( \varphi \cdot \psi \) and the process \( a.(y;b.DONE) \) does not satisfy \( \varphi \cdot \psi \). In fact, it is known from Example 4 that condition \( B \) is fulfilled, but \( D \) is not.

3. Given \( P_1 = \langle \{ x \text{ def } a.b.DONE \}, x \rangle \) and \( P_2 = \langle \{ y \text{ def } (d.b.DONE) \backslash \{ b \} \}, y \rangle \),
from \( x.a.y \), the new program:

\[ P = \langle \{ \hat{x} \text{ def } a.y; b.DONE, y \text{ def } (d.b.DONE) \backslash \{ b \} \}, \hat{x} \rangle \]
is obtained. The check of \( \hat{x} \) for the verification of the composition of formulæ \( \varphi \) and \( \psi \), that is,

\[ \varphi \cdot \psi = \langle a \rangle_0 \langle d \rangle_0 \langle b \rangle_0 \langle \top \rangle, \]
gives the result that \( a.(d.b.DONE) \backslash \{ b \}; b.DONE \) does not satisfy it. In fact, Example 4 shows that conditions \( B \) and \( D \) are fulfilled by the formulæ \( \varphi \) and \( \psi \), but \( y \) is not well-terminating.

In conclusion, given the process \( x \) satisfying the property \( \varphi \) (this proof can be performed on the transition system reduced on the basis of \( \varphi \)), the functionalities of \( x \) can be augmented through the event \( a \) and the process \( y \), provided that the condition \( B \) is respected. If the property \( \psi \) has been separately checked on the reduced transition system of \( y \), and if the result of the verification is true, the separate checking of the two properties guarantees also their composition on the basis of the chosen composition between \( x \) and \( y \). No other check is needed, apart from the verification of well-termination of \( y \) and the condition \( D \), for the case of sequential composition, and the \( \langle \omega, \rho \rangle \)-simulation for the case of parallel composition with non empty set \( L \).

The selective mu-calculus is as powerful as the mu-calculus; in addition the selective mu-calculus is particularly suited to be used in a compositional verification. The property of the selective mu-calculus, as explained in Section 2.2, is that the actions relevant for checking a formula are only those explicitly mentioned in the selective modal operators occurring in the formula itself. Due to this property, it has been possible to find the conditions \( B \) and \( D \) and the \( \langle \omega, \rho \rangle \)-simulation and to easily formulated them.

4 An example

This section describes an example scenario to illustrate the approach and to better explain Theorem 1, in particular it shows that the theorem gives only a sufficient condition. Specifically, the iterative design process of a web server is simulated: starting from the simple definition of a server that can sequentially handle only requests of static web pages from clients, some other functionalities are added to it, such as the capability of receiving data (e.g. post requests),
opening connections to, and concurrently serving requests from, multiple clients, and invoking external applications (e.g. server-side programming). For the final specification of concurrency in the system, it is chosen the process fork method illustrated in Figure 3 (the multi-threaded method could have been equivalently used). According to this model, after each request from a client, the server process is forked and the request handled by the child process. In the meanwhile, the parent process may listen to another request. After the client request has been processed, the connection is closed and the forked process expires. Note that multiple connection channels (i.e. sockets) are established, one per client, handled by different child processes.

![Client Server](image)

The CCS specification of the system has been abstracted from the HTTP protocol description. The initial situation is described by the following starting CCS program \( P = \langle \{S,T\}, S \rangle \), where:

\[
S \overset{\text{def}}{=} \text{listen}.T
\]

\[
T \overset{\text{def}}{=} \text{accept} \cdot \text{connect} \cdot \text{openCH} \cdot \text{readURL} \cdot \text{execute}.\text{writeResponse} \cdot \text{readClose} \cdot \text{closeCH}.T
\]

The CCS process \( S \) models a web server with a simplified representation. In this situation it is only important that the server is able to communicate with clients, by establishing connections, and that the request of a web page is accepted. This is described by the following formulae \( \phi_1 \) and \( \phi_2 \), respectively.

\[
S \models \phi_1 = \text{openCH}_0 \cdot \text{closeCH}_0 \cdot \text{accept}_0 \cdot \text{tt}
\]

\[
S \models \phi_2 = \text{openCH}_0 \cdot \text{readURL}_0 \cdot \text{tt}
\]
Suppose that the system is able to handle multiple connections, like in Figure 3. After the \texttt{openCH} action, a fork is applied and the set of actions of the web server is partitioned so that the child process \( F \) only handles the connection with the client and then terminates, whereas the server can listen to new connection requests from other clients. Also, the copy of the server socket of the child process is closed through the action \texttt{stopListening}. This is equivalent to starting from the process \( S' \) and augmenting it with \( F \) using the parallel composition via \texttt{openCH}, i.e. the process \( \tilde{S} \) as described below:

\[
S' \overset{\text{def}}{=} \text{listen.} T'
\]

\[
T' \overset{\text{def}}{=} \text{accept.connect.openCH.} T'
\]

\[
F \overset{\text{def}}{=} \text{stopListening.readURL.execute.writeResponse.readClose.closeCH.DONE}
\]

\[
\tilde{T} \overset{\text{def}}{=} (T' |_{\text{openCH}} F) = \text{accept.connect.openCH.} (F |_{\tilde{T}})
\]

\[
\tilde{S} \overset{\text{def}}{=} \text{listen.} \tilde{T}
\]

In this case, the requirement to be satisfied by \( \tilde{S} \) is that of concurrently handling communication with clients and accepting new connection requests. This is obtained by the formula composition \( \phi_1' |_{\text{openCH}} \psi_1 \), by Theorem 1:

\[
\tilde{S} \models \phi_1' |_{\text{openCH}} \psi_1 = (\text{openCH} \models (\text{accept} \models \text{tt})
\]

where \( \phi_1 \) and \( \psi_1 \), satisfied by \( S' \) and \( F \) respectively, are:

\[
\phi_1' = (\text{openCH} \models (\text{accept} \models \text{tt}) \text{ and;}
\]

\[
\psi_1 = (\text{closeCH} \models \text{tt})
\]

Now suppose that a new functionality is added to the system so that dynamic web pages may also be produced. The HTTP message from the client specifies, for example, the POST action. The process to be added to the server is denoted by \( F_{\text{post}} \) and the action \texttt{stopListening} is the marker action. So, the extended server definition results from applying the choice operator to \( F \), namely \( F = (F \oplus_\text{stopListening} F_{\text{post}}) \). This situation is described as follows:

\[
F_{\text{post}} \overset{\text{def}}{=} \text{readPost.executePOST.writeResponse.readClose.closeCH.DONE}
\]

\[
F \overset{\text{def}}{=} F \oplus_\text{stopListening} F_{\text{post}} = \text{stopListening.readPost.executePOST.writeResponse.readClose.closeCH.DONE} + \text{readURL.execute.writeResponse.readClose.closeCH.DONE}
\]

\[
\tilde{S} \oplus_\text{stopListening} F_{\text{post}} = \text{listen.} (\tilde{T} \oplus_\text{stopListening} F_{\text{post}})
\]

20
\[ \tilde{T} \oplus \text{stopListening } F_{\text{post}} = \text{accept.connect.openCH.}(\tilde{F} \parallel \tilde{T} \oplus \text{stopListening } F_{\text{post}}) \]

The formula composition theorem allows checking whether the server is also able to receive POST messages.

Knowing that \([ \tilde{S} \mid = \phi'_2 \mid (\text{openCH})_0 (\text{stopListening})_0 \langle \text{readURL} \rangle_0 \text{tt} \), the check is needed that \( F_{\text{post}} \) satisfies \( \psi_2 = (\langle \text{readPOST} \rangle_0 \text{tt} \) as the target formula is automatically satisfied by applying the choice operator, i.e. \( \phi'_2 \oplus \text{stopListening } \psi_2 \).

\[ \tilde{S} \oplus \text{stopListening } F_{\text{post}} \mid = \phi'_2 \oplus \text{stopListening } \psi_2 = [\text{openCH}]_0 (\text{stopListening})_0 (\langle \text{readPOST} \rangle_0 \text{tt} \lor \langle \text{readURL} \rangle_0 \text{tt} ) \]

By abuse of notation, it is called \( \tilde{S} \) the composite process \( \tilde{S} \oplus \text{stopListening } F_{\text{post}} \) and \( \tilde{T} \) the composition \( \tilde{T} \oplus \text{stopListening } F_{\text{post}} \). Some more programming on the server side can be added by invoking an external application, e.g., a DBMS. The functionality of \( \tilde{F} \), the child process, may be enlarged through the event \text{executePOST} considering a script call represented by the well-terminating finite process named \text{DBSCRIPT}. The final definition of the web server is denoted by \( WS \) and described below:

\[
\begin{align*}
\text{DBSCRIPT} & \overset{\text{def}}{=} \text{callDB.executeQuery.responseDB.DONE} \\
WS & \overset{\text{def}}{=} (\tilde{S} \text{executePOST DBSCRIPT}) = \text{listen.} \tilde{T} \text{executePOST DBSCRIPT} \\
\tilde{T} \text{executePOST DBSCRIPT} & = \text{accept.connect.openCH.} \\
& \quad (\tilde{F} \text{executePOST DBSCRIPT} \parallel \tilde{T} \text{executePOST DBSCRIPT}) \\
\tilde{F} \text{executePOST DBSCRIPT} & = \text{stopListening.} (\langle \text{readPost.executePOST.Z} \rangle_0 \langle \text{readURL.execute.L} \rangle_0 \text{tt}) \\
L & \overset{\text{def}}{=} \text{writeResponse.readClose.closeCH.DONE} \\
Z & \overset{\text{def}}{=} \text{callDB.executeQuery.responseDB.DONE; L}
\end{align*}
\]

In this case, the interest is in permitting a call to a database while executing a post script and before sending the answer to the requesting client. This results from the sequential composition \( \phi_3 \text{executePost } \psi_3 \) described below:

\[ \tilde{S} \mid = \phi_3 = [\text{executePost}]_0 (\langle \text{writeResponse} \rangle_0 \text{tt} \]

\[
\begin{align*}
\text{DBSCRIPT} & \mid = \psi_3 = (\langle \text{responseDB} \rangle_0 \text{tt} \\
WS & \mid = \phi_3 \text{executePost } \psi_3 = [\text{executePost}]_0 (\langle \text{responseDB} \rangle_0 (\langle \text{writeResponse} \rangle_0 \text{tt}}
\]

Table 4 in the Appendix reports the complete CCS description of the example.
5 Conclusion and related work

A methodology has been proposed for deducing properties, expressed by logic formulae, satisfied by systems (or programs) with modular structure through model checking of properties of the individual system modules. The paper considers systems described as CCS processes which either may represent the system specification in the design phase or can refer to system implementations, like Java programs [9, 20], or other software applications where a mapping to CCS can be automatically obtained, for example WS-BPEL processes [3].

The modules properties are verified separately, and, if satisfied, the complete system still respects them; the method is based on the syntactic structure of CCS processes and formulae. The selective mu-calculus logic [4] is employed to define modules and system properties, since this logic permits an easy property-based construction of a reduced model on which the properties can be verified. The proposed methodology is especially effective when some part (or component) of an existing system is extended with new functionalities or it is modified. Following the presented approach, the properties of the old system are reused to build up properties satisfied by the extended version of the system. The new property is obtained through the composition of the property of the old system and that of the new functionality and exploiting the required type of integration between them. The method is fully automatable.

Compositional approaches to software verification may consider decompositions of the system into modules, possibly based on the system architecture, and of the property to be individually satisfied [24], whose verification requires assumptions on the environment [1]. Other research papers in this line include the use of abstraction to extract finite state machines from C procedures [8], so to individually verify those, or else model checking software components, implemented in Java, against a high-level specification of their behavior [22]. Several other papers only consider parallel compositions, sometimes focusing on safety and liveness properties [17], and exploit techniques that preserve satisfaction of the formula [21].

Assume-guarantee reasoning first checks whether a component $M$ guarantees a property $P$, when it is a part of a system that satisfies an assumption $A$. Then, it must also be shown that the remaining components in the system satisfy $A$. Several frameworks have been proposed [21, 31] to support this style of reasoning. However, their practical impact has been limited because they require non-trivial human input in defining the assumption $A$. In Cobleigh et al. [13], Giannakopoulou et al. [19], and Pasareanu and Giannakopoulou [29], the authors present a framework to perform assume-guarantee reasoning in an incremental and fully automatic fashion, using a learning algorithm to generate and refine assumptions based on queries and counterexamples, in an iterative process. However, they can manage only safety properties. The simple compositional non-circular inference rule used in the papers cited above can be expressed...
as follows:

\[
\begin{array}{c}
\langle A \rangle M_1 \langle P \rangle \\
\langle \text{true} \rangle M_2 \langle A \rangle \\
\langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle 
\end{array}
\]

To intuitively compare this work with the previous research, consider the case when \( p \models \varphi \) and \( q \models \text{tt} \). The inference rule proposed here can be seen as a special case of the above rule where \( M_1 \) is the process \( p \), \( M_2 \) the process \( q \), \( P \) is the property \( \varphi \) and \( L = \emptyset \); the assumption \( A \) includes the conditions \( B \) and \( D \), representing the mutual influences between the two processes. It is worth noting that these conditions are syntactically based, i.e. they can be automatically deduced by the syntax of the formula and processes involved. Moreover, no limitation exists on the kind of properties. On the other hand, the approaches proposed in Cobleigh et al. [13], Giannakopoulou et al. [19], and Pasareanu and Giannakopoulou [29], can find weaker and more precise conditions being based on the counter-examples obtained by model checking the component and its environment. Note that, when \( L \neq \emptyset \), more semantic conditions are needed to ensure the soundness of the compositional technique presented in this paper. Also, this is just one aspect of the approach, as the properties of \( q \) can be exploited to derive those satisfied by the composition.

Regarding the general assume-guarantee reasoning, two points are interesting here: the circularity and the completeness of the inference rule. As shown in Namjoshi and Trefler [28], circularity is not needed, at least in principle; in fact the authors have considered the translation between circular and non-circular proofs. They argue that every compositional proof can be carried out with any of the two, at least in principle. Hence, it is shown how one can translate the application of a rule into the application of the other rule. Moreover, completeness can be reached both by circular and non circular inference rules. In particular, in Maier [25], it is shown that necessary condition for the completeness of an inference rule is the existence of side conditions, i.e. assertions exploiting variables appearing in the premises, but not in the conclusion. The inference rule presented here cannot be complete since, given a formula that the complete system satisfies, a method is not given for decomposing the formula with respect to the component processes; formula decomposition is a difficult problem that we will address in a future work. An interesting example of sound and complete inference rule is that in Amla et al. [2], which uses a methodology based on refinement (while here module composition is used) and exploits semantic assumptions among the conditions on the rules (while the methodology presented in this paper is syntax directed).

Other compositional approaches rely on tableau-based proofs to allow the dynamic generation of the system states. For example, in Dam and Gurov [14], a proof system for verifying CCS processes in the modal mu-calculus [33] is presented. To support compositional reasoning, the authors extend mu-calculus with ordinal variables, \( \kappa \), which are semantically interpreted as ordinals, and by
introducing new formulae $\mu^\kappa X.\varphi$ and $\nu^\kappa X.\varphi$, standing for the $\kappa$-th iterations in the chain of approximations to the fixed-points $\mu X.\varphi$ and $\nu X.\varphi$ respectively. Also in this case user intervention is necessary to perform the verification task in general.

The work closest to this paper is probably that by Fiser and Krishnamurthi [16], where an approach to incrementally verify collaboration-based software designs is given. Collaborations are units of software designs, encapsulating features, that can be independently developed and verified, and used to build larger designs. Each collaboration (or component) is modelled by a set of state machines and their interfaces by lists of pairs extension/re-entry points. In that model, the extension of a base component consists of sequentially adding the corresponding state/s machine/s within extension and re-entry state/s. The approach provides an incremental model checking process in that the formula that holds in the base system is only verified in the extension part, so users can deduce whether it is still valid in the extended system if some constraints are satisfied. The approach proposed in this paper, not only permits similar deductions under the given compatibility conditions, but also provides information on additional properties satisfied by the extended system.

As mentioned in the introduction, a future work can be the application of the proposed methodology to service-centric systems. Web services differ from software components in that they are stateless by nature and not physically integrated into the system but remotely invoked by it. Also, web services remain under the control and ownership of their providers. So, when building their systems, service integrators have to rely on the available descriptions of the functionality exported by these services, and possibly of their behavioral properties, while their structure will remain hidden. One of the current research issues in the web service field deals with providing (semi)automatic support to web service composition in order to correctly define the service interaction model with respect to some predefined goal [30]. In this scenario, the proposed method can be appealing as the (temporal logic) properties of the model under construction can be automatically deduced by those of the services being referred to, thus providing directions for a correct composition of the various pieces with respect to the system requirements.

References


Appendix

\[ S = \text{listen}.T \]
\[ T = \text{accept.connect} \cdot \text{readURL} \cdot \text{writeResponse} \cdot \text{readClose} \cdot \text{closeCH}.T \]

\[ S' = \text{listen}.T' \]
\[ T' = \text{accept.connect} \cdot \text{openCH}.T' \]
\[ F = \text{stopListening} \cdot \text{readURL} \cdot \text{writeResponse} \cdot \text{readClose} \cdot \text{closeCH}.DONE \]
\[ T'(F) = \text{accept.connect} \cdot \text{openCH}.(F \parallel T) \]
\[ S = \text{listen}.T \]

\[ F_{\text{post}} = \text{readPost} \cdot \text{executePOST} \cdot \text{writeResponse} \cdot \text{readClose} \cdot \text{closeCH}.DONE \]
\[ F = F \oplus \text{stopListening} \cdot F_{\text{post}} = \text{stopListening} \cdot (\text{readPost} \cdot \text{executePOST} \cdot \text{writeResponse} \cdot \text{readClose} \cdot \text{closeCH}.DONE + \text{readURL} \cdot \text{executeResponse} \cdot \text{readClose} \cdot \text{closeCH}.DONE) \]
\[ S = \text{listen}.T \]
\[ T' = \text{stopListening} \cdot F_{\text{post}} = \text{accept.connect} \cdot \text{openCH}.(F \parallel T' \oplus \text{stopListening} \cdot F_{\text{post}}) \]

\[ \text{DBSCRIPT} \overset{\text{def}}{=} \text{callDB} \cdot \text{executeQuery} \cdot \text{responseDB}.DONE \]
\[ W.S = (S \cdot \text{executePOST} \cdot \text{DBSCRIPT}) = \text{listen}.T \cdot \text{executePOST} \cdot \text{DBSCRIPT} \]
\[ T \cdot \text{executePOST} \cdot \text{DBSCRIPT} = \text{accept.connect} \cdot \text{openCH}.(F \cdot \text{executePOST} \cdot \text{DBSCRIPT}) \]
\[ F \cdot \text{executePOST} \cdot \text{DBSCRIPT} = \text{stopListening} \cdot (\text{readPost} \cdot \text{executePOST} \cdot Z + \text{readURL} \cdot \text{executeL}) \]
\[ L = \text{writeResponse} \cdot \text{readClose} \cdot \text{closeCH}.L.DONE \]
\[ E \overset{\text{def}}{=} \text{callDB} \cdot \text{executeQuery} \cdot \text{responseDB}.DONE ; E \]

Table 4. Web server specifications