

# Flexible Subtyping Relations for Component-Oriented Formalisms

Didier Buchs, Sandro Costa\*, David Hurzeler

May 29, 2002

## 1 Introduction

Because of the exponential growth of software size over the years, the major issues of maintenance and verification have become very difficult problems. One of the sub-problems of adaptability we are aiming to address in this paper is that of substitutability: given a system containing a component  $A$ , we want to be able to tell if we can “safely” replace component  $A$  by a new component  $B$ . By “safely”, we mean that we need to preserve some set of properties for the new resulting system. So we tackle the problem by first modelling the system and its components with a component/object-oriented language, and then by checking whether component  $B$ ’s model satisfies some set of properties (“type”) we might be interested in. For this, we may define an appropriate notion of *subtyping* by saying that component  $B$ ’s type is a subtype of component  $A$ ’s type iff it satisfies the set of properties mentioned above.

One of the main difficulties is to find a compromise between the kind and number of properties we want to guarantee and the practical utility of our notion of subtyping. On the one hand, strict subtyping relations can ensure significant properties over the system behaviour, but may be difficult to put into practice: in most cases, they impose restrictions we do not need, and therefore they are often hard, let alone impossible, to thoroughly check. An example of these relations is the notion of subtyping defined by Liskov [3]. On the other hand, weak relations – as the relation resulting from the application of the inheritance mechanism – can easily be applied and checked, but result in little to no guarantees over the system behaviour after substitution.

The aim of this work is to present an approach to try to work with weaker subtype relations *and* preserve important properties. The idea is to define *observers* which will describe the way the system – or a set of clients – interacts with a particular component. With this proposal, we can use subtyping relations that take into account *observable*<sup>1</sup> and/or expected properties of components – the components are dealt with as black boxes with some defined points of interaction (for instance, methods). The observer accounts for the context of use of a component. Our idea here is to define a subtype relation by an observer

---

\*also from the Universidade Federal da Paraíba, Campina Grande - Brazil

<sup>1</sup>Observable properties of a component are properties we (the system or a set of clients) know of this component without seeing its internal state.

and a set of properties. We may therefore have as many subtype relations as we want, thus making the definition flexible and practical. Substitutability, in our opinion, depends on the context of use and the set of properties we are interested in preserving.

The problem was first encountered using the CO-OPN (Concurrent Object-Oriented Petri Nets) [1] specification language, which is basically an object-oriented formalism allowing for concurrency, distribution, and synchronizations between objects. We decided to formally define our notion of subtyping and observation on a more general component-oriented formalism, so as to enlarge its scope of use.

The first part of this paper is the formal definition of our component-oriented formalism. It is important to note that the aim here is not to build models of a system, but rather to study properties of given components and systems. We define the notion of observational subtyping in the second part of the paper, and then conclude and explain the work still to be done in the last part. We will illustrate our discussion along the way with a small buffer example.

## 2 Definitions and Notations for a General Formalism

In this section we give some notations and definitions relative to the general object-oriented formalism in which we will be working.

### 2.1 Concepts

**Components** A *component* is considered as an independent entity composed of an *internal state* and which provides some *services* to the exterior. The only way to interact with an object is to ask for its services; the internal state is then protected against uncontrolled accesses, but services provided by an object may alter its internal state. Our point of view is that *encapsulation* is an essential feature of object-orientation and there should be no way of violating it. The call of one of a component's services is called an *event*.

**Component Composition** An *component composition* is characterized as an arrangement of a set of components together with the way they can interact. In our approach, two components can interact if there is, at least, one way to synchronize their elements (available and required services). We say that a component composition defines a new component, that can be rearranged with other components to form a new one, and so on. Sometimes, we prefer to call a component a *system of components*.

**Concurrency** In our formalism, each component possesses its own behaviour and concurrently evolves with the others.

**Component Identity** Each component has an identity, which is also called a component identifier, and may be used as a reference. Moreover, once the kind of properties which interest us and the context (in terms of the other components of a system) are given, a type (set of properties) is explicitly associated with each component.

## 2.2 Syntax and Semantics of Data Structures

In specifications, algebraic abstract data types are adopted for the description of values. In this section, we will define the syntax and semantic of data structures as a way to describe data values, and see how we can use these specifications to define the algebras on which our components' states will be defined. So the aim of the first section is simply to define what we call a  $\Sigma$ -algebra.

### 2.2.1 Syntax of the data structures

For the following definitions, we consider an universe which includes the two following disjoint sets:  $\mathbf{S}$ , the set of all sort names and  $\mathbf{F}$ , the set of all function names.

**Definition 2.1 (S-Sorted Set)** *Let  $S \subseteq \mathbf{S}$  be a finite set. A  $S$ -sorted set  $A$  is a disjoint union of a family of sets indexed by  $S$  ( $A = \bigcup_{s \in S} A_s$ ), noted as  $A = (A_s)_{s \in S}$ .*

**Definition 2.2 (Signature)** *A signature is a couple  $\Sigma = \langle S, F \rangle$ , where  $S \subseteq \mathbf{S}$  is a finite set of sorts and  $F = (F_{w,s})_{w \in \mathbf{S}^*, s \in \mathbf{S}}$  is a  $(\mathbf{S}^* \times \mathbf{S})$ -sorted set of function names of  $\mathbf{F}$ . Each  $f \in F_{\emptyset, s}$  is called a constant.*

Let us give an example. Let  $S$  be  $\{\text{Naturals}\}$ .

Then  $A = (A_{\text{Naturals}}) = (0, 1, 2, \dots)$ . We may then consider the addition function:  $"+" \in F_{\text{Naturals Naturals, Naturals}}$ .

**Definition 2.3 (Terms of a Signature)** *Let  $\Sigma = \langle S, F \rangle$  be a signature and  $X$  be a  $S$ -sorted set of variables. The set of terms of  $\Sigma$  over  $X$  is a  $S$ -sorted set  $T_{\Sigma, X}$ , where each set  $(T_{\Sigma, X})_s$  is inductively defined as follows:*

- each variable  $x \in X_s$  is a term with the sort  $s$ , i.e.,  $x \in (T_{\Sigma, X})_s$
- each constant  $f \in F_{\emptyset, s}$  is a term with the sort  $s$ , i.e.,  $f \in (T_{\Sigma, X})_s$
- for all operations that are not a constant  $f \in F_{w,s}$ , with  $w = s_1 \dots s_n$ , and for all  $n$ -tuple of terms  $(t_1 \dots t_n)$  such that all  $t_i \in (T_{\Sigma, X})_{s_i}$  ( $1 \leq i \leq n$ ),  $f(t_1 \dots t_n) \in (T_{\Sigma, X})_s$

**Definition 2.4 (Axioms)** *Let  $\Sigma = \langle S, F \rangle$  be a signature and  $X$  be a  $S$ -sorted set of variables. The axioms are equations<sup>2</sup>  $t = t'$  such that  $t, t' \in (T_{\Sigma, X})_s$ ,  $s \in S$ , are terms with the same sort.*

**Definition 2.5 (algebraic specification)** *A many sorted algebraic specification  $\text{spec} = \langle S, F, AX \rangle$  is a signature extended by a collection of axioms  $AX$ .*

### 2.2.2 Semantics of the data structures

For this section, let  $\Sigma = \langle S, F \rangle$  be a complete signature.

**Definition 2.6 ( $\Sigma$ -Algebra)** *A  $\Sigma$ -algebra is a couple  $A = \langle D, O \rangle$ , in which  $D$  is a  $S$ -sorted set of values ( $D = D_{s_1} \cup \dots \cup D_{s_n}$ ) and  $O$  is a set of functions, such that for each function name  $f \in F_{w,s}$ ,  $w = s_1 \dots s_n$  there is a function  $f^A \in O$ , defined as  $f^A : D_{s_1} \times \dots \times D_{s_n} \rightarrow D_s$ .*

<sup>2</sup>This is a simplified version of the Horn Clauses.

**Definition 2.7 (Interpretation of Variables)** Let  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra and  $X$  be a  $S$ -sorted set of variables. An interpretation from  $X$  into  $D$  is a function  $\sigma : X \rightarrow D$ , such that, for each  $s \in S$ , if  $x \in X_s$  then  $\sigma(x) \in D_s$ .

**Definition 2.8 (Evaluation of a Term)** Let  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra and  $X$  be a  $S$ -sorted set of variables. An evaluation of a term  $t \in T_{\Sigma, X}$  w.r.t. an interpretation  $\sigma$ , written as  $\llbracket t \rrbracket_{\sigma}^A$ , is inductively defined as follows:

- if  $t$  is a variable then  $\llbracket t \rrbracket_{\sigma}^A = \sigma(t)$
- if  $t$  is a constant, i.e.  $t$  is in the form  $f$ , then  $\llbracket t \rrbracket_{\sigma}^A = f^A$
- if  $t$  is in the form  $f(t_1 \dots t_n)$ , where  $t_1 \dots t_n$  are terms and  $f \in F$ , then  $\llbracket t \rrbracket_{\sigma}^A = f^A(\llbracket t_1 \rrbracket_{\sigma}^A \dots \llbracket t_n \rrbracket_{\sigma}^A)$ .

**Definition 2.9 (Validity of an Equation)** Let  $X$  be a  $S$ -sorted set of variables and  $t, t' \in T_{\Sigma, X}$  be two terms. An equation  $t = t'$  is valid in a  $\Sigma$ -algebra  $A$  if and only if, for any interpretation of variables  $\sigma$ ,  $\llbracket t \rrbracket_{\sigma}^A = \llbracket t' \rrbracket_{\sigma}^A$ .

**Definition 2.10 (Model)** A  $\Sigma$ -algebra  $A$  is a model to a spec  $= \langle S, F, AX \rangle$  if and only if all equations in  $AX$  are valid in  $A$ .

## 2.3 Component Model

In this section, we present the formal definition of our component model syntax and semantics, as well as the semantics of operations such as component composition.

### 2.3.1 Components

The syntactic part of the components is composed by an *interface* and a *set of attributes*.

The notion of component interface supports the description of the component entities that are responsible for the communication with the external environment. These entities are the *available methods* – services – and the *gates* – required services.

**Definition 2.11 (Interface)** Let  $\Sigma = \langle S, F \rangle$  be a signature and  $\mathcal{C}$  be a set of component names. An interface, based on both a signature  $\Sigma$  and the set  $\mathcal{C}$ , is a tuple  $I = \langle M, G \rangle$ , in which  $M$  is a  $\mathcal{C} \times S^*$ -sorted set of method names and  $G$  is a  $\mathcal{C} \times S^*$ -sorted set of gate names.

The following definition of set *Events* is later used to characterize the *local events* of an component.

**Definition 2.12 (Events)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names and  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names. The set of events  $Events_{A, \mathcal{C}, M, G}$  can be recursively built together with the set  $GateExpr_{A, \mathcal{C}, G}$ . They are the least sets such that:

- $\forall o \in \mathcal{C}, \forall g \in G_{o, s_1, \dots, s_n}, \forall d_i \in (D)_{s_i}, i \in 1, \dots, n,$   
 $o.g(d_1, \dots, d_n) \in GateExpr_{A, \mathcal{C}, G}$

- $\forall x, x' \in GateExpr_{A,C,G}, \forall op \in \{\|, \dots, \oplus\}, x \text{ op } x' \in GateExpr_{A,C,G}$
- $\forall o \in C, \forall m \in M_{o,s_1,\dots,s_n}, \forall d_i \in (D)_{s_i}, i \in 1, \dots, n, o.m(d_1, \dots, d_n) \in Events_{A,C,M,G}$
- $\forall o \in C, \forall m \in M_{o,s_1,\dots,s_n}, \forall d_i \in (D)_{s_i}, i \in 1, \dots, n, \forall x \in GateExpr_{A,C,G}, o.m(d_1, \dots, d_n) \textbf{ with } x \in Events_{A,C,M,G}$
- $\forall e, e' \in Events_{A,C,M,G}, \forall op \in \{\|, \dots, \oplus\}, e \text{ op } e' \in Events_{A,C,M,G}$

Then, a local event can be described as (i) a simple method call – for example,  $o_1.m_1(a_1, a_2) -$ , (ii) a synchronisation between a method call and an expression composed by gate names – for example,  $o_1.m_2(a_1, a_2) \textbf{ with } o_1.g_1(a_1)$  or  $(o_1.m_2(a_1, a_2) \textbf{ with } o_1.g_1(a_1)) \| o_1.g_2(a_2)$  – or (iii) a complex expression – for example,  $o_1.m_1(a_1, a_2) \oplus (o_1.m_2(a_1, a_2) \textbf{ with } o_1.g_1(a_1))$ . The expressions composed by gate names and complex expressions of events are built with the operators:

- “ $\|$ ” for the occurrence in parallel (in the Petri net sense),
- “ $\dots$ ” for the occurrence in sequence, and
- “ $\oplus$ ” for the non-deterministic occurrence.

Occurrence in parallel “in the Petri net sense” refers to simultaneous occurrence of events with simultaneous access to resources, and where each event may not use resources simultaneously created by the other one(s). We can explain the intended meaning of event expressions as follows. The **with** synchronisation links a furnished service (a method) to a set of required services (gates). For example, the observable event “ $o_1.m_2(a_1, a_2) \textbf{ with } o_1.g_1(a_1) \| o_1.g_2(a_2)$ ” represents the parallel synchronization of the method  $m_2$  of a component  $o_1$  with both gates  $g_1$  and  $g_2$  of this same component. There are two simultaneous required services calls when the  $o_1.m_2$  service is called.

Next, we define set *States* of all possible states of a set of attributes. Set *States* is useful in the description of component behaviour.

**Definition 2.13 (States of a Set of Attributes)** *Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  a  $\Sigma$ -algebra and  $B$  a  $S$ -sorted set of attributes. The set of states  $States_A(B)$  of set  $B$  is built as follows:*

- $\epsilon \in States_A(B)$  represents the absence of attributes.
- $\forall s \in S, \forall b \in B_s, \forall d \in D_s, (b, d) \in States_A(B)$  represents the presence of an element in an attribute  $b$ .
- $\forall e, e' \in States_A(B), e \| e' \in States_A(B)$  represents the availability of two sets of resources simultaneously in a state.

**Definition 2.14 (State Equivalence)** *State equivalence, noted  $\equiv$ , is the equivalence relation on  $States_A(B)$  defined by:*

- $\forall e, e', e'' \in States_A(B), e \equiv e' \Rightarrow e \| e'' \equiv e' \| e''$
- $\forall e, e', e'' \in States_A(B), (e \| e') \| e'' \equiv e \| (e' \| e'')$ .

Please note that in what follows, by making an abuse in the notations, we shall write  $States_A(B)$  instead of  $States_A(B)/\equiv$ . Also note that here, the  $\parallel$  symbol does not have the same meaning as before: It is now simultaneous availability of resources, whereas it meant simultaneous occurrence of event in the previous context. In what follows, this symbol's signification will depend on its context.

A property resulting from the above definition is that  $\parallel$  preserves the associativity in  $States_A(B)$ , i.e,  $\forall e, e', e'' \in States_A(B), (e \parallel e') \parallel e'' \equiv e \parallel (e' \parallel e'')$ .

**Example 2.1** *The set of states  $States_A(B)$  related to an algebra  $A$  with the set of values restricted to one possible value  $Blacktoken = \{\text{@}\}$  and the set of attributes  $B$  restricted to just one attribute  $B = \{b\}$  of sort  $Blacktoken$  is:*

$$States_A(B) = \{\epsilon, (b, \text{@}), (b, \text{@}) \parallel (b, \text{@}), (b, \text{@}) \parallel (b, \text{@}) \parallel (b, \text{@}), \dots\}$$

At this point, we have defined all the needed elements to express what we mean by a *component*.

**Definition 2.15 (Components)** *Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra and  $\mathcal{C}$  be a set of component names. A component is a tuple  $\langle c, I_c, B_c, Trans_c \rangle$ , in which:*

- $c \in \mathcal{C}$  is its name,
- $I_c = \langle M_c, G_c \rangle$  is its interface, s.t.  $M_c$  and  $G_c$  are two  $\{c\} \times S^*$ -sorted sets,
- $B_c$  is a  $S$ -sorted set of attribute names,
- $Trans_c \subseteq States_A(B_c) \times Events_{A, \{c\}, M_c, G_c} \times States_A(B_c)$  is the labelled transition system that defines the component semantics. A triple  $\langle s, e, s' \rangle \in Trans_c$  is noted  $s \xrightarrow{e} s'$ .

We can see in Figure 1 the illustration of a component  $c_1$ , whose interface is composed of methods  $m_1$  and  $m_2$  and gates  $g_1$  and  $g_2$ . We use black and white rectangles to illustrate respectively methods and gates.

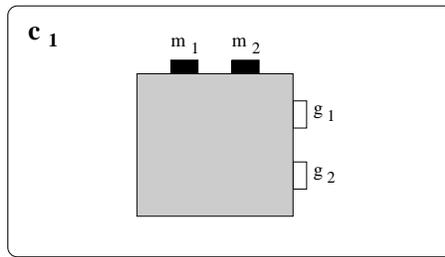


Figure 1: Illustration of a component with two methods and two gates

In Figure 2, we can see a detail of an internal synchronization of component  $c_1$  (in fact, a possible local event):  $m_1$  **with**  $g_1 \parallel g_2$ . This internal synchronization is shown in the component semantics depicted in Figure 3.

However, transition system  $Trans_{c_1}$  does not contain the complete semantics of component  $c_1$ . It is necessary to complete it in order to cope with

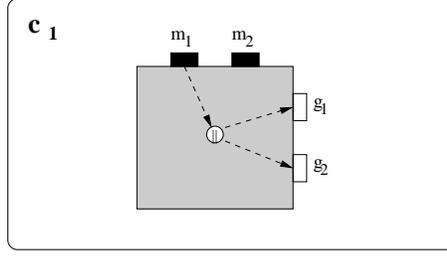


Figure 2: The method  $m_1$  are synchronised with the occurrence in parallel of the gates  $g_1$  and  $g_2$ .

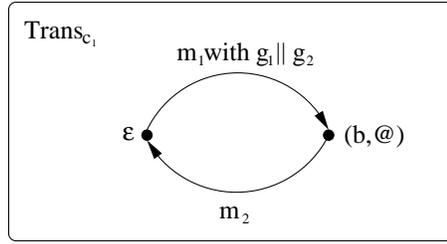


Figure 3: Labelled transition system of the component  $o_1$ , illustrated in Figure 2.

both the multisets of states –  $(b, @) \parallel (b, @) \xrightarrow{m_2} (b, @)$ , for example – and the occurrence in parallel ( $\parallel$ ), in sequence ( $\cdot$ ) and non-deterministic ( $\oplus$ ) of events –  $(b, @) \parallel (b, @) \xrightarrow{m_2} \varepsilon$ , for example. For coping with multisets of states, we define the *CompMono* operation that completes a transition system with the semantics related to multisets. We also define the operation *Closure* that completes a transition system with the transitions related to events in sequence, in parallel and non-deterministic.

The reason why we have chosen not to put the complete semantics of the component in its definition (in  $Trans_{c_1}$ ) is that implicitly, the semantics of the absorption of resources is seen as an extension of the basic semantics of methods and gates.

**Definition 2.16 (CompMono)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $B$  be a  $S$ -sorted set of attributes,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names,  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names and  $Trans \subseteq States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  be a labelled transition system.

$CompMono_{A,B} : States_A(B) \times Events_{A,C,M,G} \times States_A(B) \mapsto States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  is s.t:

$CompMono_{A,B}(Trans)$  is the labelled transition system inductively defined as follows:

$$\frac{s \xrightarrow{a} s' \in Trans}{s \xrightarrow{a} s' \in CompMono_{A,B}(Trans)} \text{ inclusion}$$

$$\frac{s \xrightarrow{a} s' \in Trans, t \in States_A(B)}{s || t \xrightarrow{a} s' || t \in ComMono_{A,B}(Trans)} \text{ multisets}$$

In Figure 4, we can see the result of the application of  $CompMono$  on the transition system  $Trans_{o_1}$  depicted in Figure 3.

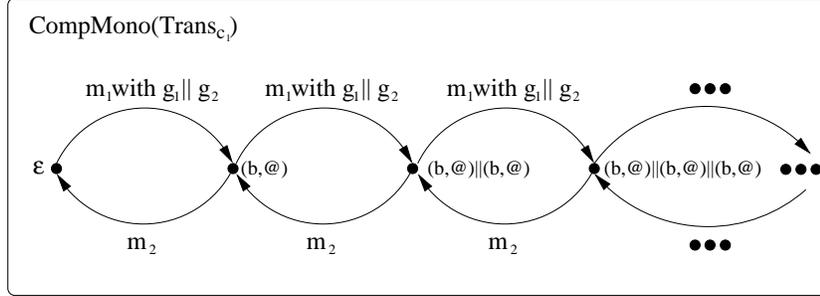


Figure 4:  $CompMono(Trans_{c_1})$ .

**Definition 2.17 (Closure)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $B$  be a  $S$ -sorted set of attributes,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names,  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names and  $Trans \subseteq States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  be a labelled transition system.

$Closure : States_A(B) \times Events_{A,C,M,G} \times States_A(B) \mapsto States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  is s.t:

$Closure(Trans)$  is inductively defined as follows:

$$\frac{s \xrightarrow{a} s' \in Trans}{s \xrightarrow{a} s' \in Closure(Trans)} \text{ include}$$

$$\frac{s \xrightarrow{a} s' \in Closure(Trans), s' \xrightarrow{b} s'' \in Closure(Trans)}{s \xrightarrow{a..b} s'' \in Closure(Trans)} \text{ seq}$$

$$\frac{s \xrightarrow{a} s' \in Closure(Trans), t \xrightarrow{b} t' \in Closure(Trans)}{s || t \xrightarrow{a || b} s' || t' \in Closure(Trans)} \text{ paral}$$

$$\frac{s \xrightarrow{a} s' \in Closure(Trans)}{s \xrightarrow{a \oplus b} s' \in Closure(Trans)} \text{ alt1}$$

$$\frac{s \xrightarrow{b} s' \in Closure(Trans)}{s \xrightarrow{a \oplus b} s' \in Closure(Trans)} \text{ alt2}$$

For example, the application of *Closure* on the transition system  $T_{c_1}$  (Figure 3) extends it with the inclusion of arcs  $(b, @) \xrightarrow{m_2 \cdot (m_1 \text{ with } g_1 \parallel g_2)} (b, @)$  and  $(b, @) \parallel (b, @) \xrightarrow{m_2 \parallel m_2} \varepsilon$ , among others.

Then, with this two operations, we can define the complete semantics of a component. As a remark, let us just point out the fact that operators *Compmono* and *Closure* are both idempotent.

**Definition 2.18 (Complete Semantics of a Component)** *Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra and  $\langle c, I_c, B_c, Trans_c \rangle$  be a component.*

*The complete semantics of the component  $c$  is the transition system  $Sem_c \subseteq States_A(B_c) \times Events_{A, \{c\}, M_c, G_c} \times States_A(B_c)$ , defined as:*

$$Sem_c = Closure(CompMono(Trans_c))$$

Let us now see an example (in Figure 5). Let  $B_1$  be a buffer with a capacity of two. Let us assume that the producer only produces one kind of unit: @. The buffer has two methods *put* and *get*, two gates *failput* and *failget* called if the buffer cannot provide the put and get services, and one attribute *o* containing the units. The dashed arrows represent internal synchronizations. Figure 5 also depicts a part of  $Trans_{b_1}$ . In the semantics shown, all states are shown (as the buffer's capacity is of two), but there are in fact infinitely more arcs (some of which have been represented using a dashed arrow). For instance, we have an arc from state  $(-, -)$  to state  $(@, @)$  labelled by  $put \parallel (put \parallel (put \text{ with } failput))$ .

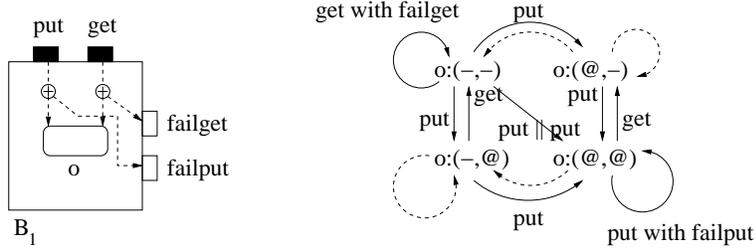


Figure 5: The buffer example and part of its semantics

### 2.3.2 Component Composition

In this section, we cope with the creation of new components through the composition of other pre-existing ones. This operation requires, beyond the set of components to be composed, a new interface (in fact, a restriction on the union of the interfaces of the basic components) and the way the components are connected. In the language we define in this work, to connect components means to synchronize some of their interface elements. Then, some component outputs (gates) are synchronized with some component inputs (methods), and this is described by the means of *synchronization expressions*.

**Definition 2.19 (Synchronization Expressions)** *Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names and  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names.*

*Sets  $ExprSync_{A, \mathcal{C}, M, G}$  and  $MethodExpr_{A, \mathcal{C}, M}$  are the least sets such that:*

- $\forall o \in \mathcal{C}, \forall m \in M_{o,s_1,\dots,s_n}, \forall d_i \in (D)_{s_i}, i \in 1, \dots, n,$   
 $o.m(d_1, \dots, d_n) \in MethodExpr_{A,C,M}$
- $\forall x, x' \in MethodExpr_{A,C,M}, \forall op \in \{\|, \dots, \oplus\},$   
 $x op x' \in MethodExpr_{A,C,M}$
- $\forall o \in \mathcal{C}, \forall g \in G_{o,s_1,\dots,s_n}, \forall d_i \in (D)_{s_i}, i \in 1, \dots, n, \forall x \in MethodExpr_{A,C,M},$   
 $o_i.g(d_1, \dots, d_n) \mathbf{with} x \in ExprSync_{A,C,M,G}$

$ExprSync_{A,C,M,G}$  is called the set of synchronisation expressions.

Note that in what follows, given  $e \in ExprSync_{A,C,M,G}$  we will note  $Gates_e$  the set of gates appearing on the left of the **with** in the elements of  $e$ .

The idea is that a synchronization expression connects a component gate to a set of component methods, by the means of the **with** synchronization. The set of operators  $\{\|, \dots, \oplus\}$  is used to construct the method expressions  $MethodExpr$  in the same way it is used to construct  $GateExpr$  in the definition of  $Events$ .

As an example, in Figure 6 gives an example with a system composed by two components  $c_1$  and  $c_2$ , connected by the set of expressions  $Expr = \{c_1.g_1 \mathbf{with} c_2.m_3, c_1.g_2 \mathbf{with} c_2.m_4\}$ . Then, gate  $c_1.g_1$  may be synchronized<sup>3</sup> with the method  $c_2.m_3$ , as well as  $c_1.g_2$  and  $c_2.m_4$ . To make this composition complete, we would need also to define a new interface.

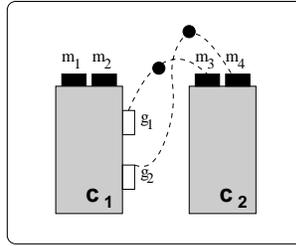


Figure 6: The composition of the two components  $c_1$  and  $c_2$  through the synchronization expressions  $c_1.g_1 \mathbf{with} c_2.m_3$  and  $c_1.g_2 \mathbf{with} c_2.m_4$ .

In order to obtain the semantics of the new component, we have to apply a sequence of operations on the semantics of each of its composing components. In Figure 7, we find an illustration of this process.

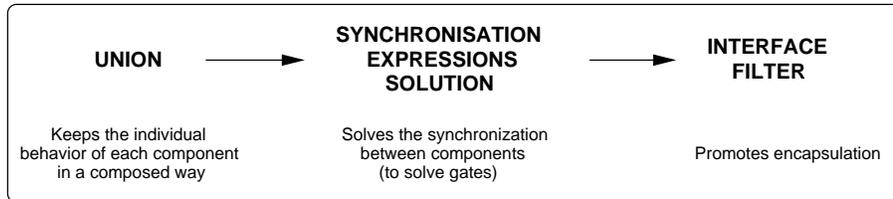


Figure 7: Steps to obtain the semantics of a composed component

<sup>3</sup>We allow that distinct expressions can be associated to the same gate.

## Union

With the *union* of the semantics of each component in the composition, we aim to compose an initial transition system (noted as  $\rightarrow_U$ ) that takes into account the combination of the composing components states.

**Definition 2.20 (Transition Systems Union)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $Trans = \{Trans_1, \dots, Trans_n\}$  be a set of labelled transition systems and for each transition system  $Trans_i \subseteq States_A(B_i) \times E_i \times States_A(B_i)$ ,  $B_i$  is a set of attributes and  $E_i$  is a set of labels.

The union of the transition systems elements of  $Trans$ , which we shall note  $\biguplus_{1 \leq i \leq n} Trans_i \subseteq States_A(\bigcup_{1 \leq i \leq n} B_i) \times \bigcup_{1 \leq i \leq n} E_i \times States_A(\bigcup_{1 \leq i \leq n} B_i)$  is a labelled transition system, s.t.:

$$\biguplus_{1 \leq i \leq n} Trans_i = CompMono(\bigcup_{1 \leq i \leq n} Trans_i)$$

For example, the Figure 9 shows the transition system obtained by the union of the transitions  $Trans_{c_2}$  and  $Trans_{c_3}$ , illustrated in the Figure 8.

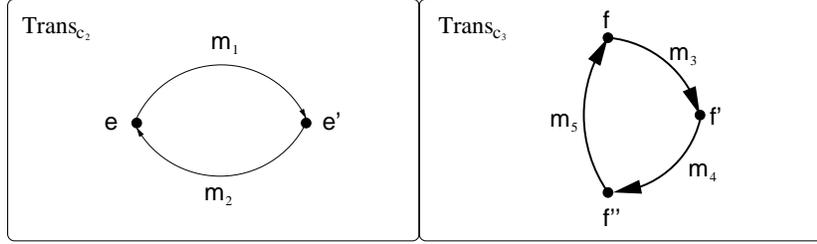


Figure 8: The transition systems of two components  $c_2$  and  $c_3$ .

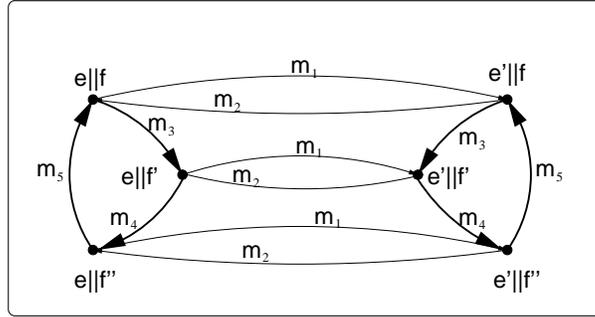


Figure 9: The union of  $Trans_{c_2}$  and  $Trans_{c_3}$ .

## Synchronization Expressions Solution

The goal of the second step, the *synchronization expressions solution*, is to add the transitions relative to the synchronisation expressions. The idea is to solve the links between the gates and the methods described through the expressions.

Intuitively, we have to reflect the fact that the effects of a synchronized event can also be assigned to the *initial event*, i.e., to the event that starts the synchronization (a method). As a result, we obtain a transition system (noted as  $\rightarrow_S$ ) that defines the internal behaviour of the new component.

We use a set of gradual examples to properly explain this step. For each new example, we show a more complete inference rule for the new transitions to be added. To simplify things, we consider just fragments of a toy example.

First of all, we have to consider that each transition in  $\rightarrow_U$  is also in  $\rightarrow_S$ , i.e.:

$$\frac{s \xrightarrow{a}^U s'}{s \xrightarrow{a}^S s'} \text{ include}$$

Therefore, the autonomy of the basic components is preserved in the new one.

The first example, used to explain the general idea, is showed in Figure 10<sup>4</sup>. In this example, we have a system composed by two components,  $c_1$  and  $c_2$  that are connected by the expression  $\langle g_1 \text{ with } m_3 \rangle$ , i.e., gate  $g_1$  of component  $c_1$  *may be* synchronized with method  $m_3$  of component  $c_2$ . Moreover, the picture shows that, internally to component  $c_1$ , gate  $g_1$  is synchronized with the occurrence of method  $m_1$ <sup>5</sup>. Then, we have to include into the system semantics a transition which label is the initial event  $m_1$  and the effect is the same of the eventual parallelism between the synchronisation  $\langle m_1 \text{ with } g_1 \rangle$  and the method  $m_3$ . In the Figure 11, we can see such a process, applied to some fragments of the  $o_1$  and  $o_2$  behaviours. The idea is that an occurrence of  $m_1$  can hide a synchronisation with  $m_3$ .

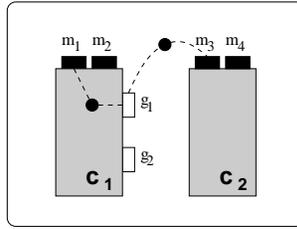


Figure 10: Composition of components  $c_1$  and  $c_2$

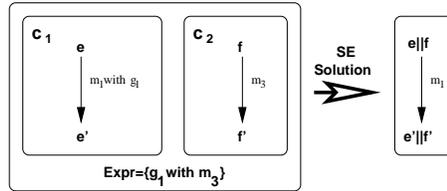


Figure 11: Synchronization solution

<sup>4</sup>To simplify, we avoid writing  $o.m$  when refer to method  $m$  of component  $o$  when the context is clear. For coherence reasons, each method and gate in the examples have a unique name in the system.

<sup>5</sup>There is no behaviour associated just to gates. So, every gate is internally linked to at least a method.

We can now write the following inference rule:

$$\frac{s \stackrel{m}{\longrightarrow}_S \text{with } g \ s', t \stackrel{m'}{\longrightarrow}_S t', g \text{ with } m' \in Expr}{s \parallel t \stackrel{m}{\longrightarrow}_S s' \parallel t'} \text{ simplegatesolutions}$$

However, this rule is not enough to handle all the potential synchronization occurrences, that can involve more complex expressions.

For example, in the Figure 12, we can distinguish two synchronization expressions,  $\langle g_1 \text{ with } m_3 \rangle$  and  $\langle g_2 \text{ with } m_4 \rangle$  related to two gates,  $g_1$  and  $g_2$ , that are – in component  $c_1$  – synchronized with the same method:  $m_1$ . Then, an occurrence of  $m_1$  can hide a synchronization with both  $m_3$  and  $m_4$ . To obtain the proper result for this synchronization, we need a way to relate the expressions  $g_1 \parallel g_2$  and  $m_3 \parallel m_4$ .

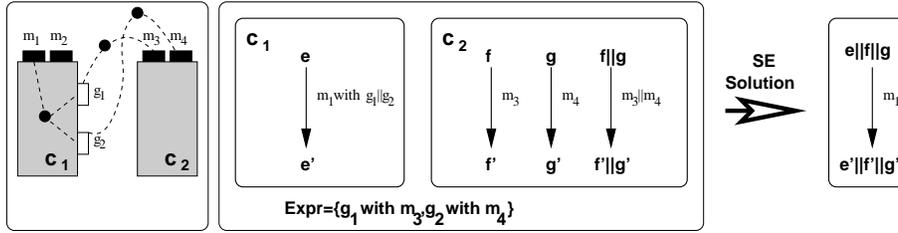


Figure 12: More complex synchronization solution

We could use a hypothetical relation  $SyncEvSub_{Expr,G,M}$  that would associate a gate expression on  $G$  to a method expression on  $M$  w.r.t. the set of synchronization expressions  $Expr$ . In the example of Figure 12, we would have  $\langle g_1 \parallel g_2, m_3 \parallel m_4 \rangle \in SyncEvSub_{Expr,G,M}$ .

An inference rule that would take into account the relation  $SyncEvSub_{Expr,G,M}$  could be:

$$\frac{s \stackrel{m}{\longrightarrow}_S \text{with } x \ s', t \stackrel{y}{\longrightarrow}_S t', \langle x, y \rangle \in SyncEvSub_{Expr,G,M}}{s \parallel t \stackrel{m}{\longrightarrow}_S s' \parallel t'} \text{ incompletegatesolution}$$

So, for the example of Figure 12, we would have the following deduction:

$$\frac{s \stackrel{m_1}{\longrightarrow}_S \text{with } g_1 \parallel g_2 \ s', t \stackrel{m_3 \parallel m_4}{\longrightarrow}_S t', \langle g_1 \parallel g_2, m_3 \parallel m_4 \rangle \in SyncEvSub_{Expr,G,M}}{s \parallel t \stackrel{m_1}{\longrightarrow}_S s' \parallel t'}$$

The problem is that, sometimes, the compatibility between a gate expression  $x$  and a method expression  $y$  is not total. As an example, in Figure 12, expressions  $g_1 \parallel g_2$  and  $m_3$  are partially compatible, because from them and the synchronization expressions  $Expr$  we can only solve gate  $g_1$ . Gate  $g_2$  can then be solved in the future by another synchronization expression, either internally, w.r.t. the composition between components  $c_1$  and  $c_2$ , or externally, with other components. Then, to solve such a kind of partial compatibility between expressions, we have to take into account the *remainder gate expression*, i.e., the

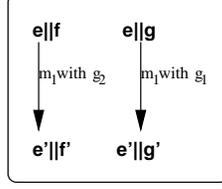


Figure 13: Remainder gate expression example 1

residue of a solution operation ( $g_2$  in the example above). The effect of the partial solution of that example is illustrated in Figure 13.

Another example of a remainder gate expression is depicted in Figure 14. In this new example, we have a synchronization expression,  $\langle g_1 \textbf{with} m_3 || m_4 \rangle$ , that connects gate  $g_1$  to method  $m_3$  (together with method  $m_4$ ), which is internally synchronized with gate  $g_3$ . Then, when we solve the link  $\langle g_1 \rightarrow m_3 \rangle$ , we have to synchronize the method  $m_1$  with the gate  $g_3$ . So, the gate expression  $g_1$  is partially compatible with the method expression  $(m_3 \textbf{with} g_3) || m_4$  and the remainder of the solution is the expression  $g_3$ .

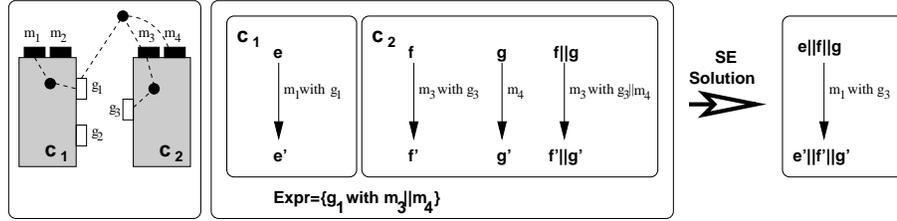


Figure 14: Remainder gate expression example 2

So, we have to extend relation  $SyncEvSub$  in order to cope with the remainder gate expression. The actual relation  $SyncEvSub$  links the gate expressions, the event expressions and the remainder gate expressions. For the example of Figure 12, we have  $\langle g_1 || g_2, m_3, g_2 \rangle, \langle g_1 || g_2, m_4, g_1 \rangle \in SyncEvSub$ . For the example of Figure 14, we have  $\langle g_1, (m_3 \textbf{with} g_3) || m_4, g_3 \rangle \in SyncEvSub$ .

Before we present the definition of relation  $SyncEvSub$ , we need to define the auxiliary relation  $AllSyncEvSub$ . This relation links three elements: a method expression, a compatible event expression and the proper remainder gate expression. For example,  $\langle m_3 || m_4, (m_3 \textbf{with} g_3) || m_4, g_3 \rangle \in AllSyncEvSub$ .

**Definition 2.21 (Relation  $AllSyncEvSub$ )** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names and  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names.

$AllSyncEvSub_{A, \mathcal{C}, M, G} \subseteq MethodExpr_{A, \mathcal{C}, M} \times Events_{A, \mathcal{C}, M, G} \times GateExpr_{A, \mathcal{C}, G}$  is inductively defined below, bearing in mind that we note  $x \leftarrow y / z$  for  $\langle x, y, z \rangle$  in the relation:

$$\frac{c \in \mathcal{C}, m \in M_{c, s_1, \dots, s_n}, d_i \in (D)_{s_i}, i \in 1, \dots, n}{c.m(d_1, \dots, d_n) \leftarrow c.m(d_1, \dots, d_n) / \varepsilon} \text{ simplemethodsynchro}$$

$$\frac{c \in \mathcal{C}, m \in M_{c,s_1,\dots,s_n}, d_i \in (D)_{s_i}, i \in 1, \dots, n, z \in \text{GateExpr}_{A,\mathcal{C},G}}{c.m(d_1, \dots, d_n) \leftarrow c.m(d_1, \dots, d_n) \textbf{ with } z/z} \text{ synchmeth}$$

$$\frac{x \leftarrow y/z, x' \leftarrow y'/z', z \neq \varepsilon, z' \neq \varepsilon, op \in \{\|, \dots, \oplus\}}{x \text{ op } x' \leftarrow y \text{ op } y'/z \text{ op } z'} \text{ compop}$$

$$\frac{x \leftarrow y/z, x' \leftarrow y'/\varepsilon, op \in \{\|, \dots, \oplus\}}{x \text{ op } x' \leftarrow y \text{ op } y'/z, x' \text{ op } x \leftarrow y' \text{ op } y/z} \text{ compemptyop}$$

Briefly, the first rule means that simple synchronization does not produce any residue. The second rule implies that a method with an unresolved synchronization will keep the latter as residue. The third describes how to compose two tuples of this relation, and so does the last one, but where one of the components has no residue.

We now need a specialisation of this relation in order to cope with the *Expr* variable in the composition.

**Definition 2.22 (Relation *SyncEvSub*)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names,  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names and  $\text{Expr} \subseteq \text{ExprSync}_{A,\mathcal{C},M,G}$  be a set of synchronisation expressions.

$\text{SyncEvSub}_{A,\mathcal{C},M,G,\text{Expr}} \subseteq \text{GateExpr}_{A,\mathcal{C},G} \times \text{Events}_{A,\mathcal{C},M,G} \times \text{GateExpr}_{A,\mathcal{C},G}$  is inductively defined below. As above, we will note  $x \leftarrow y/z$  for  $\langle x, y, z \rangle$  in the relation.

$$\frac{g \textbf{ with } x \in \text{Expr}, x \leftarrow y/z}{g \leftarrow y/z} \text{ sync}$$

$$\frac{v \leftarrow y/z, v' \leftarrow y'/z', z \neq \varepsilon, z' \neq \varepsilon, op \in \{\|, \dots, \oplus\}}{v \text{ op } v' \leftarrow y \text{ op } y'/z \text{ op } z'} \text{ compop}$$

$$\frac{v \leftarrow y/z, v' \leftarrow y'/\varepsilon, op \in \{\|, \dots, \oplus\}}{v \text{ op } v' \leftarrow y \text{ op } y'/z, v' \text{ op } v \leftarrow y' \text{ op } y/z} \text{ compemptyop}$$

$$\frac{v \leftarrow y/z, z \neq \varepsilon, op \in \{\|, \dots, \oplus\}, v' \in \text{GateExpr}_{A,\mathcal{C},G}}{v \text{ op } v' \leftarrow y/z \text{ op } v', v' \text{ op } v \leftarrow y/v' \text{ op } z} \text{ enrich}$$

$$\frac{v \leftarrow y/\varepsilon, op \in \{\|, \dots, \oplus\}, v' \in \text{GateExpr}_{A,\mathcal{C},G}}{v \text{ op } v' \leftarrow y/v' \cup v' \text{ op } v \leftarrow y/v'} \text{ emptyenrich}$$

The *AllSyncEvSub* relation describes all the potential arcs in the labelled transition system resulting from the composition of several components. Then, the *SyncEvSub* relation picks among these all the possible resolutions from synchronizations specified in the *Expr* synchronization expression. These two relations are in fact syntactic, and it is now that the semantics come into play, with the complete synchronization expression solution definition.

**Definition 2.23 (Synchronization Expressions Solution)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C}$  be a set of component names,  $M$  be a  $\mathcal{C} \times S^*$ -sorted set of method names,  $G$  be a  $\mathcal{C} \times S^*$ -sorted set of gate names,  $B$  be a set of attributes,  $Sem \subseteq States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  be a labelled transition system (noted  $\rightarrow_U$ ) and  $Expr \subseteq ExprSync_{A,C,M,G}$  be a set of synchronisation expressions.

$SESolution : ExprSync_{A,C,M,G} \times States_A(B) \times Events_{A,C,M,G} \times States_A(B) \mapsto States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  is s.t.:

$SESolution_{Expr}(Sem)$  is inductively defined as follows:

$$\frac{s \xrightarrow{a}_U s'}{s \xrightarrow{a}_S s'} \text{ include}$$

$$\frac{s \xrightarrow{m}_{S} \text{with } x \ s', t \xrightarrow{y}_S t', x \Leftarrow y/z}{s \parallel t \xrightarrow{m}_{S} \text{with } z \ s' \parallel t'} \text{ syncwithrest}$$

$$\frac{s \xrightarrow{m}_{S} \text{with } x \ s', t \xrightarrow{y}_S t', x \Leftarrow y/\varepsilon}{s \parallel t \xrightarrow{m}_S s' \parallel t'} \text{ syncwithoutrest}$$

The first rule states that the synchronizations specified in  $Expr$  have to be included in the relation. The other rules are interpreted very much as in the relation  $AllSyncEvSub$ , only now the base has changed a little according to  $Expr$ .

### Interface Filter

The objective of the last step in the process of the component composition semantics construction, the *interface filter*, is to improve the encapsulation. This is done by eliminating or hiding the behaviour associated to the elements not in the new interface. Then, this operation's result is a new transition system, noted  $\rightarrow_F$ , resulting from the filter of  $\rightarrow_S$  w.r.t. the system interface.

**Definition 2.24 (Interface Filter)** Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C}$  be a set of component names,  $I = \langle M, G \rangle$  and  $I_{new} = \langle M_{new}, G_{new} \rangle$  be two interfaces based on  $\Sigma$  and  $\mathcal{C}$ , such that  $M_{new} \subseteq M$  and  $G_{new} \subseteq G$ ,  $B$  be a set of attributes and  $Sem \subseteq States_A(B) \times Events_{A,C,M,G} \times States_A(B)$  be a labelled transition system (noted  $\rightarrow_S$ ).

$IFilter_{I_{new}} : States_A(B) \times Events_{A,C,M,G} \times States_A(B) \mapsto States_A(B) \times Events_{A,C,M_{new},G_{new}} \times States_A(B)$ , called the filter of  $Sem$  w.r.t the new interface  $I_{new}$ , is s.t.:

$IFilter_{I_{new}}(Sem)$  (which transforms  $\rightarrow_S$  into  $\rightarrow_F$ ) is inductively defined as follows:

$$\frac{s \xrightarrow{o.m(d_1, \dots, d_n)}_S s', m \in M_{new}}{s \xrightarrow{o.m(d_1, \dots, d_n)}_F s'} \text{ incl}$$

$$\frac{s \xrightarrow{o.m(d_1, \dots, d_n)}_S \text{with } x \quad s', m \in M_{new}, x \in GateExpr_{A,C,G_{new}} \quad gatekeep}{s \xrightarrow{o.m(d_1, \dots, d_n)}_F \text{with } x \quad s'}$$

Let us give an example of this. Suppose we have two arcs, labelled by  $c_1.m_1, c_1.m_2$  **with**  $c_1.g_1$  and  $c_1.m_2$  **with**  $c_1.g_2$  in the labelled transition system we are considering. Let  $I = \{c_1.m_2, c_1.g_1\}$  be a new interface. Now suppose we want to apply  $I\text{Filter}_I$  to the labelled transition system presented. The resulting labelled transition system will have a single arc labelled by  $c_1.m_2$  **with**  $c_1.g_1$ .

Finally, we can define the *component composition*:

**Definition 2.25 (Component Composition)** *Let  $\Sigma = \langle S, F \rangle$  be a signature,  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra,  $\mathcal{C} = \{\langle c_1, I_{c_1}, B_{c_1}, Trans_{c_1} \rangle, \dots, \langle c_n, I_{c_n}, B_{c_n}, Trans_{c_n} \rangle\}$  be a set of components, in which  $I_{c_i} = \langle M_{c_i}, G_{c_i} \rangle$  is the interface of the component  $c_i$  and  $M = \bigcup_{1 \leq i \leq n} M_{c_i}$  and  $G = \bigcup_{1 \leq i \leq n} G_{c_i}$  are respectively the sets of all method and gate names in  $\mathcal{C}$ ,  $I_{new} = \langle M_{new}, G_{new} \rangle$  be an interface, such that  $M_{new} \subseteq M$  and  $G_{new} \subseteq G$ ,  $Expr \subseteq ExprSync_{A,C,M,G}$  be a set of synchronisation expressions<sup>6</sup> and  $c$  be a component name.*

*The composition of the set of components  $\mathcal{C}$  w.r.t. the interface  $I_{new}$ , the set of synchronisation expressions  $Expr$  and the identifier  $o$  is the component  $ObjComp(\mathcal{C}, I_{new}, Expr, c) = \langle c, I_c, B_c, Trans_c \rangle$ , such that:*

- $I_c = I_{new}$
- $B_c = B_{c_1} \cup \dots \cup B_{c_n}$
- $Trans_c = I\text{Filter}_{I_{new}}(SE\text{Solution}_{Expr}(\biguplus_{1 \leq i \leq n} Sem_{c_i}))$

Note that we also use the following notation in what follows:  $s \circ_{I, Expr} t = ObjComp(\{s, t\}, I, Expr, c)$ . An example of composition is described in Figure 15: we compose the buffer  $B_1$  described above with another identical buffer  $B_2$ . The idea is that we can then have a buffer with a capacity of four. The interface of the new buffer is  $\{put1, get1, failput2, failget2\}$ , the set of synchronization expressions is  $\{failput1$  **with**  $put2, failget1$  **with**  $get2\}$ , so that when the first buffer cannot provide the asked service, it relays to the second buffer (but in a hidden way).

### 3 Observational Subtyping

In this section, we will define our notion of typing and subtyping. We will first give some basic definitions and define what we call the type of a component, then define our general notion of subtyping; After that, we will give a practical particular case of subtype relation and finish with the buffer example.

<sup>6</sup>We make a small abuse of notations and also note  $\mathcal{C}$  the set of component names of  $\mathcal{C}$

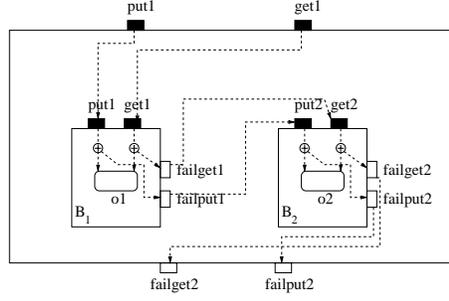


Figure 15: An example of composition of two buffers

### 3.1 Properties contexts and types

The main issue here is property preservation. So our notion of typing highly depends on the kind of properties we are interested in. Therefore, we first define what we call a *properties context*. Please note that we will use the notations defined in the previous sections. We will also use the buffer example illustrated in Figure 5.

**Definition 3.1 (Properties context)** *Let  $C$  be a set of components. Let  $(V_C, \vee, \wedge)$  be a ring. Let us call the elements of  $V_C$  validation formulae. Let  $\models_C: C \times V_C \mapsto \{true, false\}$  be a function called the satisfaction relation. We call Properties context a tuple  $\langle C, V_C, \models_C \rangle$ .*

Intuitively,  $V_C$  represents the set of all possible properties of the kind we are interested in for a set of components  $C$ . An examples of such a set of properties might be  $HML_\infty$  (the Hennessy-Milner logic [4] extended to the infinite branching case [8][9]) if we use branching based equivalence relation. It might also simply be interface inclusion if we are only interested in the method names in the interface.

In the buffer example, the tuple  $\langle B_1, HML_\infty(B_1), \models_{B_1} \rangle$  is a possible properties context, with  $HML_\infty(B_1)$  the set of properties on  $B_1$  described by the logic mentioned above, and  $\models_{B_1}$  the natural satisfaction relation that goes with it.

We may compose these properties contexts:

**Definition 3.2 (Properties context composition)** *Let  $\Sigma = \langle S, F \rangle$  be a signature and  $A = \langle D, O \rangle$  be a  $\Sigma$ -algebra. Let  $\langle C_1, V_{C_1}, \models_{C_1} \rangle$  and  $\langle C_2, V_{C_2}, \models_{C_2} \rangle$  be two properties contexts. Let  $M$  and  $G$  respectively be the set of all method and gate names in  $C_1 \cup C_2$ . Let  $C_3$  be a new name,  $Expr \in ExprSync_{A, C_1 \cup C_2, M, G}$  a set of synchronization expressions, and  $I \subseteq M \cup G$  a new interface. The composition of the two properties contexts defined above with respect to  $C_3, Expr$ , and  $I$  is the properties context  $\langle CompObj(C_1 \cup C_2, I, Expr, C_3), V_{C_3}, \models_{C_3} \rangle$ , where:*

- $(V_{C_3}, \vee, \wedge)$  is the smallest ring containing  $V_{C_1} \cup V_{C_2}$ .
- $\models_{C_3}: C_3 \times V_{C_3} \mapsto \{true, false\}$ .

This composition gives us a relation between  $V_{C_1 \cup C_2}$  and  $V_{C_1}$  and  $V_{C_2}$ : The former is the union of all elements of the latters, and of all conjunctions and

disjunctions of elements of these lattices. Also please note that we have chosen to have a general definition by not giving more detail on  $\models_{C_3}$ . So satisfaction of the new properties context restricted to one of the components may not be the same as satisfaction on the component alone (for instance if we are interested in comparative properties with other components): If a part of  $\langle C_1, V_{C_1}, \models_{C_1} \rangle$  satisfies a property, it may not do so anymore in  $\langle C_3, V_{C_3}, \models_{C_3} \rangle$ .

Coming back to the buffer example, let us examine the component resulting from the composition of two buffers, as depicted in Figure 15. The new buffer  $B_3$  now allows us to define a new properties context  $\langle B_3, HML_\infty(B_3), \models_{B_3} \rangle$ . As we have just noted,  $\models_{B_3}$  restricted to the two composing buffers is not  $\models_{B_1}$  or  $\models_{B_2}$ : Indeed,  $put1..put1$  is now fireable.

We may now define what we call the type of a component. In what follows, we assume that we have a function  $C \mapsto V_C$  which, given a set of components, gives us the set of all the properties of these components we might be interested in. The *Attrib* function is also implicitly assumed to be given. We also use the notations defined above.

**Definition 3.3 (Type of a component)** *Let  $pc = \langle C, V_C, \models_C \rangle$  be a properties context and  $c \in C$  be a component. The type of  $c$  w.r.t. the properties context  $pc$  is defined by:  $type_{pc}(c) = \{vf \in V_C / c \models_C vf\}$ .*

Something which results from our definition is that the type of a component depends on its context. We feel that the properties of a component may also depend on the system it is embedded in. We also define an equivalence relation on these types:

**Definition 3.4 (Type equivalence)** *Let  $pc = \langle C, V_C, \models_C \rangle$  be a properties context and  $a, b \in C$  be two components. Type equivalence w.r.t.  $pc$  (noted  $\equiv_{pc}$ ) between components  $a$  and  $b$  is defined by:*

$$(a \equiv_{pc} b) \iff (type_{pc}(a) = type_{pc}(b)) \iff (\forall vf \in V_C, a \models_C vf \iff b \models_C vf).$$

As an example, if we have chosen  $HML_\infty$  [8][9] as properties set, type equivalence would be bisimulation (cf [10]). Conversely, if we have chosen simple interface inclusion, then type equivalence is simply interface equality (see Figure 16 below for an illustration)

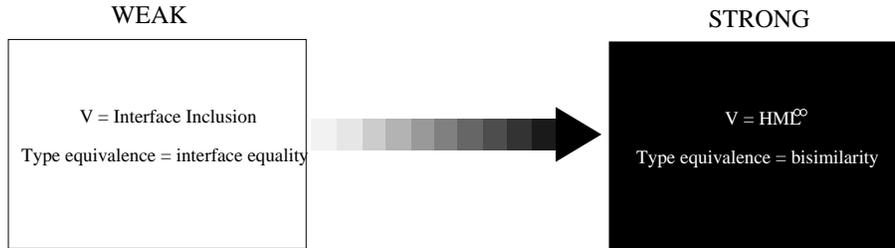


Figure 16: Type equivalence strength w.r.t. the set of properties chosen

### 3.2 General subtype relation: Definition and properties

We are now ready to define subtyping in our component formalism. Let us first remind us that our problem is that of substitutability: we want to know if we may “safely” replace a component  $t$  by another component  $s$  in a system containing the set of components  $C$ . “Safely” means that we want some properties ( $vf \in V_C$ ) to be preserved. But we also want to take into account the way the system used to interact with “old” component  $t$ . So we are in fact interested in proving that some sub-behaviour of  $s$  satisfies the properties  $vf$ .

We will first define what we call an *observation context*:

**Definition 3.5 (Observation context)** *An observation context is a tuple  $\langle \Sigma, A, pc, \{t, obs_t\}, Expr, I \rangle$  s.t.:*

- $\Sigma = \langle S, F \rangle$  a signature and  $A = \langle D, O \rangle$  a  $\Sigma$ -algebra
- $pc = \langle C, V_C, \models_C \rangle$  a properties context.
- $t, obs_t \in C$  two components, with  $t = \langle t, \{M_t, G_t\}, B_t, Trans_t \rangle$  and  $obs_t = \langle obs_t, I_{obs_t}, B_{obs_t}, Trans_{obs_t} \rangle$ ,
- $M = M_{obs_t} \cup M_t$  and  $G = G_{obs_t} \cup G_t$ .
- $Expr \in ExprSynC_{A, \{obs_t, t\}, M, G}$  a synchronization expression and  $I \subseteq I_{obs_t}$  an interface.

This observation context represents the elements on which we are going to define subtyping: A set of possible properties, a set of components, and two particular components and the way we compose them (via  $Expr$  and  $I$ ). We are now ready to define the general notion of subtyping:

**Definition 3.6 (Subtyping w.r.t. an observation context and a property)**

*Let us consider:*

- $ObsCont_t = \langle \Sigma, A, pc, \{t, obs_t\}, Expr, I \rangle$  an observation context with  $pc = \langle C, V_C, \models_C \rangle$ ,  $t = \langle t, I_t, B_t, Trans_t \rangle$ ,
- $s = \langle s, I_s, B_s, Trans_s \rangle \in C$  a component s.t.  $I_t \subseteq I_s$ .
- $vf \in V_C$  a property (which may be the conjunction of a set of properties).

*Subtyping of  $s$  w.r.t.  $t$ ,  $ObsCont_t$ , and  $f$ , (noted  $s \preceq_{ObsCont_t, f} t$ ) holds iff:*

$$(s \circ_{I, Expr} obs_t) \models_C vf \Leftrightarrow (t \circ_{I, Expr} obs_t) \models_C vf.$$

*We then say that  $s$  is a subtype of  $t$  or that it is substitutable for  $t$  w.r.t. to observation context  $ObsCont_t$  and properties  $f$ .*

So we see here that our notion of subtyping has another component ( $obs_t$ ) as parameter (via the observation context). This parameter’s first role is to filter behaviours on which we do *not* need the  $f$  properties to be preserved: we constrain the component behaviour to check, and then evaluate a validation formula on the resulting behaviour. The idea is to interact only with the observer component during the latter evaluation. As a result, we reduce the semantics to check. This parameter may also add behaviours, and may finally filter some of the feed-back calls (gate calls) from the component to check. Our aim is to make the subtype relation as flexible as possible.

Note that the validation formula may express a property (or set of properties) for a single observation context, as well as for several observation contexts (for instance if we are comparing them). Moreover, it may express property on the several observer components.

We have the following coherence result:

**Result 1 (Coherence result)** *Let us consider:*

- $pc = \langle C, V_C, \models_C \rangle$  a properties context,
- $ObsCont_a = \langle \Sigma, A, pc, \{a, obs_a\}, Expr, I \rangle$  an observation context,
- $b, c, d \in C$  be three components,
- $vf \in V_C$  a validation formula.

We then have:

$$(b \preceq_{ObsCont_a, vf} a) \wedge (a \equiv_{pc} c) \wedge (b \equiv_{pc} d) \iff (d \preceq_{ObsCont_a, vf} c)$$

We will now make some remarks on these subtype relations. First of all, this subtype relation seems to be symmetrical; this is *not* the case. Indeed, the observation context and more specifically the observer component depends on the supertype component and not on the subtype component.

Second, we can try to see if our subtype relation satisfies Hameurlain's two requirements [7]: Compositionality and transitivity. The point here is that we *can* satisfy them if we want to.

First of all, when the observation context is restrictive (meaning that, when composed with a subtype component, it actually restricts its original semantics), we get some form of compositionality which fulfills the first requirement.

The second requirement is about transitivity. Now, as such, our relation is not transitive: It is not a preorder relation. The reason for this is that we are interested in a very particular problem, where we want to replace a component  $s_1$  by a new one  $s_2$ . We are not interested in the problem where we want to re-replace this new component  $s_2$  by yet another one  $s_3$ . Or if we are, we treat it in a whole new context, because we feel that the problem environment has changed: We are not interested in how the system interacted with the original component  $s_1$ , but in how it now interacts with  $s_2$ . This makes us want to work with another observation context (adapted to the new substitutability problem) and so we are interested in a different subtyping relation. The reason for our not having any transitivity is that we define one subtyping relation per substitution context. However, we may write, keeping the same notations as above:

**Result 2 (Pseudo-transitivity)** *Let  $pc = \langle C, V_C, \models_C \rangle$  be a properties context, and  $ObsCont_a = \langle \Sigma, A, pc, \{a, obs_a\}, Expr, I \rangle$  an observation context. Let  $b, c \in C$  be some components,  $vf \in V_C$  a validation formula. We then have:*

$$(b \preceq_{pc, ObsCont_a, vf} a) \wedge (c \preceq_{pc, ObsCont_a, vf} b) \implies (c \preceq_{pc, ObsCont_a, vf} a)$$

This is only “pseudo-transitivity” as in the left part of the implication, we have built an observer and observation context according to another component  $a$  in  $c \preceq_{C, ObsCont_a, vf} b$ .

Let us now define a form a hierarchy between these subtype relations.

**Definition 3.7 (Subtype strength relation)** *Let us consider*

- $pc = \langle C, V_C, \models_C \rangle$  a properties context.
- $ObsCont1_t = \langle \Sigma, A, pc, \{t, obs1_t\}, Expr1, I \rangle$  and  $ObsCont2_t = \langle \Sigma, A, pc, \{t, obs2_t\}, Expr2, I \rangle$  two observation contexts
- $f1, f2 \in V_C$  two properties.

We say that  $\preceq_{ObsCont1_t, f1}$  is stronger than  $\preceq_{ObsCont1_t, f1}$  iff:

$$Trans_{obs2_t, t} \subseteq Trans_{obs1_t, t} \wedge ((f1 = f2) \vee (\exists f \in V_C / (f1 = f2 \wedge f) \vee (f2 = f1 \vee f)))$$

**Result 3 (Subtype strength relation property)** *The above relation is a pre-order relation.*

The idea here is that a subtype relation is stronger than an other one if its properties or its associated observation context are more constraining.

We have now given our global definition of subtyping, and some global properties it satisfies. Let us now see a practical application of this definition.

### 3.3 Exterior system subtype relation

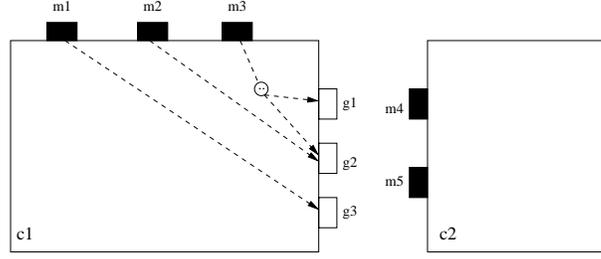
We will now give a particular example of observer component and observation context, which filters any method call<sup>8</sup> outside from the exterior world semantics exactly. The situation is this: we want to replace a component  $c$  in a system  $totSys$  which is the result of a composition of  $c$  with a  $RestSys$  component (representing the rest of the system). The idea is to build an observer which we compose with a new candidate for  $c$ , and which will “simulate”  $RestSys$  in the sense that it will filter any behaviour  $RestSys$  could not have had. To do this, we first need to define two operators on gate and method expressions. The idea is to build operators which transpose the behavioural restriction on the transition system of  $RestSys$  to the transition system of  $c$  with which it is composed. We will explain our definitions on the example illustrated in Figure 17 below. In this example, we will only depict part of the complete labelled transition system of  $c1$ .

**Definition 3.8 (HideFilter operator w.r.t. an observation context)** *Let  $ObsCont_c = \langle \Sigma, A, pc, \{c, RestSys\}, Expr, I \rangle$  be an observation context. Then the operator  $HideFilter_{ObsCont_c} : GateExpr_{A, RestSys} \mapsto GateExpr_{A, RestSys}$  is defined by:*

- $HideFilter_{ObsCont_c}(g) = g$  if  $g \in Gates_{Expr}$ ,
- $HideFilter_{ObsCont_c}(g) = \emptyset$  if  $g \notin Gates_{Expr}$ ,
- $HideFilter_{ObsCont_c}(g \text{ op } \tilde{g}) = g \text{ op } HideFilter_{ObsCont_c}(\tilde{g})$  if  $g \in Gates_{Expr}$ ,  $op \in \{+, \dots, \parallel\}$ ,
- $HideFilter_{ObsCont_c}(g \text{ op } \tilde{g}) = HideFilter_{ObsCont_c}(\tilde{g})$  if  $g \notin Gates_{Expr}$ ,  $op \in \{+, \dots, \parallel\}$ ,

<sup>7</sup> where we note:  $obs1_t.t = t \circ_{I, Expr1} obs1_t$  and  $obs2_t.t = t \circ_{I, Expr2} obs2_t$ .

<sup>8</sup> Please note that we may also build an observer which filters method calls from the exterior world and gate calls from the observed component



$$\text{Expr}=\{g1 \text{ with } m4, g3 \text{ with } (m4..m5)\}$$

Figure 17: Example of two components to compose and a synchronization expression

What this operator does is simply “hide” on the labels of a labelled transition system’s arcs gate names which are not in a given synchronization expression *Expr*. Let us give an example. Figure 18 below gives on the example of Figure 17 part of the complete labelled transition system of *c1* before and after the application of this operator on its labels.

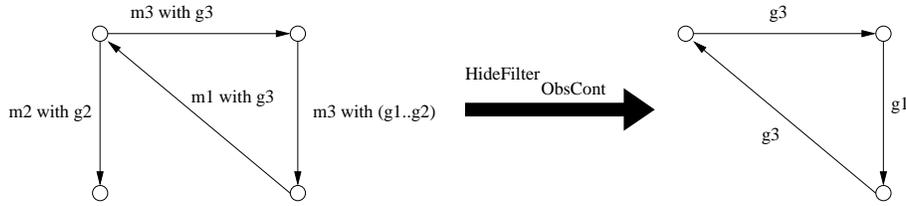


Figure 18: Example of application of operator *HideFilter* on part of a labelled transition system

So this first operator places us from the point of view of the events sent by a component (via the gates). Let us now see how this effects the transition system of the component which receives these event calls.

**Definition 3.9 (SubsFilter operator w.r.t. an observation context)** *Let  $\text{ObsCont}_c = \langle \Sigma, A, pc, \{c, \text{RestSys}\}, \text{Expr}, I \rangle$  be an observation context. Then the operator  $\text{SubsFilter}_{\text{ObsCont}_c} : \text{HideFilter}_{\text{ObsCont}_c}(\text{GateExpr}_{A, \text{RestSys}}) \mapsto \text{MethodExpr}_{A, c}$  is defined by:*

- $\text{SubsFilter}_{\text{ObsCont}_c}(g) = \tilde{m}$  if  $g$  **with**  $\tilde{m} \in \text{Expr}$ ,
- $\text{SubsFilter}_{\text{ObsCont}_c}(g) = \emptyset$  else,
- $\text{SubsFilter}_{\text{ObsCont}_c}(g \text{ op } \tilde{g}) = \text{SubsFilter}_{\text{ObsCont}_c}(g) \text{ op } \text{SubsFilter}_{\text{ObsCont}_c}(\tilde{g})$ ,  
 $\text{op} \in \{+, \dots, \parallel\}$ .

We show the effect of this operator on the same example. Figure 19 shows the result of applying this operator to the result of applying the *HideFilter* operator on part of the complete semantics of *c1*. So we see that it simply replaces

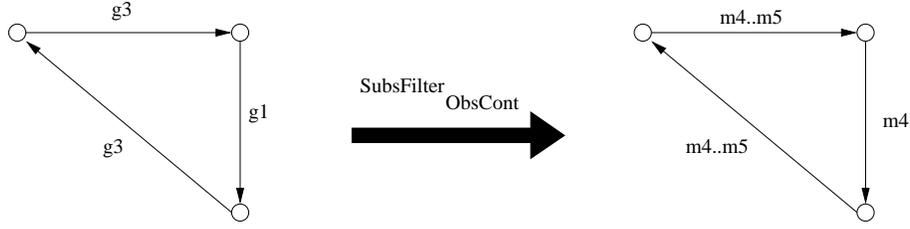


Figure 19: Example of application of operator SubsFilter on part of a labelled transition system already treated by HideFilter

the gates present in the synchronization expression  $Expr$  by the corresponding method calls in  $c2$ .

We are now ready to define the exterior system observation context and the exterior system subtype relation.

**Definition 3.10 (Exterior system observation context)** Let  $ObsCont_c = \langle \Sigma, A, pc, \{c, RestSys\}, Expr, I \rangle$  be an observation context. An exterior system observation context w.r.t.  $ObsCont_c$  is an observation context  $ObsCont_{c_{ext}} = \langle \Sigma, A, pc, \{c, obs\}, Expr', I' \rangle$  where  $obs = \langle obs, \langle M_{obs}, G_{obs} \rangle, B_{obs}, Trans_{obs} \rangle$  is s.t.:

- $I_{obs} \subseteq I_c$ ,
- $B_{obs} = B_{RestSys}$ ,
- $Trans_{obs}$  is the smallest set inductively defined by:

$$\frac{s \xrightarrow{\bar{m} \text{ with } \bar{g}} s'' \in Trans_{RestSys}}{s \xrightarrow{SubsFilter_{ObsCont_c} (HideFilter_{ObsCont_c} (\bar{g}))} s' \in Trans_{obs}} \text{extsys}trans$$

Let us now explain the idea behind this definition. Taking our example back again, we will define an observer, which, according to Figure 19, may only fire  $m4..m5$ , then only  $m4$ , and then  $m4..m5$  and so on. This observer, composed to  $c2$  (in this case) simulates the calls  $c1$  might do to  $c2$  by filtering any other behaviour. This is symbolically shown in Figure 20, where we have designed an observer having the expected behavioural restriction via a Petri net. Here we have not chosen to have  $B_{obs} = B_{RestSys}$  for readability reasons. In Figure 21, we show how the observer component is connected to  $c2$ .

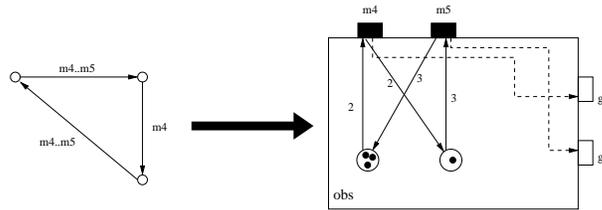
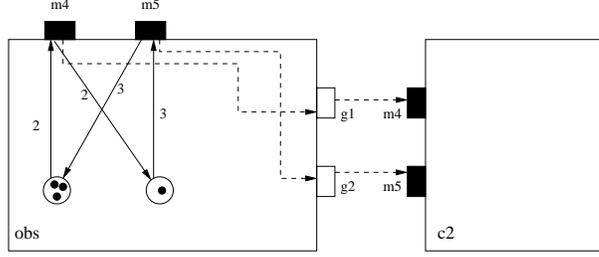


Figure 20: Exterior system observer for  $c2$  according to behavioural constraints

A small note here: Building an observer from a semantics or behavioural restriction is typically non-trivial. Some work is being conducted on this part, but obviously, in the general case, this problem is undecidable. It may be impossible to know how the rest of the system exactly interacts with the component, but we can always make an approximation which guarantees that we will obtain a sub-transition system which keeps enough (in the sense defined by our substitution purposes) properties of the original system. Note that observation makes verification easier: the labelled transition system on which we must verify a property is only part of the component's original labelled transition system.



$$\text{Expr} = \{g1 \text{ with } m4, g3 \text{ with } (m4..m5)\}$$

Figure 21: Connected exterior system observer for  $c2$

The reason why the method and gate names of the observer are also found in component  $c$  is that, as we will see, in the subtype relation defined after that, the composition of the observer with  $c$  and with the subtype component is such that every method name in the observer is synchronized with the same method name in the component, and the same goes for gates. This is how the observer actually restricts the component's behaviour.

So we also define the notion of exterior system subtype relation:

**Definition 3.11 (Exterior system subtype relation)** *Let us consider:*

- $ObsCont_t = \langle \Sigma, A, pc, \{t, RestSys\}, Expr, I \rangle$  an observation context with  $pc = \langle C, V_C, \models_C \rangle$ ,  $t = \langle t, I_t, B_t, Trans_t \rangle$
- $ObsCont_{t.out} = \langle \Sigma, A, pc, \{t, obs_t\}, Expr', I' \rangle$  an exterior system observation context w.r.t.  $ObsCont_t$  s.t.  $Expr' = \{obs_t.m \text{ with } t.m\} \cup \{obs_t.g \text{ with } t.g\}$  (where  $m$  is a method name and  $g$  is a gate name).
- $s = \langle s, I_s, B_s, Trans_s \rangle \in C$  a component s.t.  $I_t \subseteq I_s$ .
- $vf \in V_C$  a property (which may be the conjunction of a set of properties).

Then, the subtyping of  $s$  w.r.t.  $t$ ,  $ObsCont_{t.out}$ , and  $vf$  is called exterior system subtyping w.r.t.  $t$ ,  $ObsCont_t$ ,  $ObsCont_{t.out}$ , and  $vf$ .

So we want to substitute a new component for an old one in a system  $totSys = ObjComp(\{t, RestSys\}, I, Expr)$ . We know how the (rest of the) system  $RestSys$  used to interact with the old one  $t$ , and we know which properties we are interested in preserving. So this subtype relation corresponds to substitutability for this context of use and for these properties.

We may also want to prove properties on a restriction of the semantics describing the way  $RestSys$  used to interact with  $t$ . This is what we will do now.

In what follows, given  $\langle c, I_c, B_c, Trans_c \rangle$  be a component and  $T \subseteq Trans_c$  a sub-behaviour of  $c$ , we note  $c/T$  the component  $\langle c/T, I_c, B_c, T \rangle$ .

Then, using the same notations as above, the (trivial) property which validates our notion of subtyping is:

**Result 4 (Substitutability)** *Let us consider:*

- $ObsCont_t = \langle \Sigma, A, pc, \{t, RestSys/T\}, Expr, I \rangle$  an observation context with  $pc = \langle C, V_C, \models_C \rangle$ ,  $t = \langle t, I_t, B_t, Trans_t \rangle$ , and  $T \subseteq Trans_{RestSys}$  is some sub-behaviour of  $RestSys$ ,
- $ObsCont_{t_{ext}} = \langle \Sigma, A, pc, \{t, obs_t\}, Expr', I' \rangle$  an exterior system observation context w.r.t.  $ObsCont_t$  s.t.  $Expr' = \{obs_t.m \textbf{ with } t.m\} \cup \{obs_t.g \textbf{ with } t.g\}$
- $s = \langle s, I_s, B_s, Trans_s \rangle \in C$  a component s.t.  $I_t \subseteq I_s$ .
- $vf \in V_C$  a property (which may be the conjunction of a set of properties).

Then we say that  $s$  is substitutable for  $t$  w.r.t. to properties  $vf$  and the context of use  $T$  iff

$$(s \circ_{I, Expr} obs_t) \models_C vf \Leftrightarrow (t \circ_{I, Expr} RestSys/T) \models_C vf$$

In other words, in that case, subtyping is equivalent to substitutability.

### 3.4 Back to the buffer example

Our problem is the following: we want to know if we may substitute a new buffer for the old buffer in a given system (producer + consumer + buffer) when:

- the rest of the system (producer + consumer) alternatively calls the *put* and the *get* method of the buffer.
- we are only interested in the property: “if the rest of the system puts an element in the buffer, it wants to be able to get it back some time in the future”.

So we first build the observer component accordingly to the behavioural restrictions given. This is shown in Figure 22. Thus, we create an observer component which filters behaviours other than alternative calls of *put* and *get* (symbolized by a Petri net on the picture).

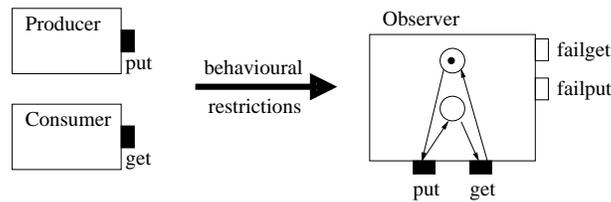


Figure 22: Observer for the buffer example

What we call *Observation* of a component  $c$  with its observer  $obs$  is interaction with the component  $observed_c = c \circ_{I_{obs}, SyncExpr} obs$  where  $SyncExpr \in ExprSync_{A, \{c, obs\}, M_{obs} \cup M_c, G_{obs} \cup G_c}$  is s.t.:

$$SyncExpr = \{obs.m \textbf{ with } c.m, m \in M_{obs}\} \cup \{c.g \textbf{ with } obs.g, g \in G_{obs}\}$$

This is shown in Figure 23.

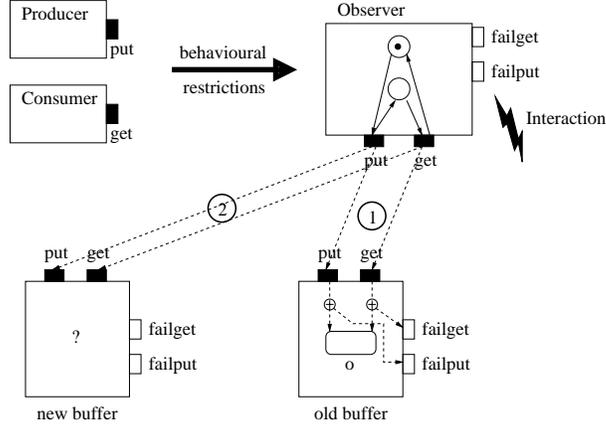


Figure 23: Observation for the buffer example

Finally, we test the substitutability of the new buffer w.r.t. the old one. Testing the validation formula on the buffer composed with the observer is sufficient according to our third Result. This is shown in Figure 24.

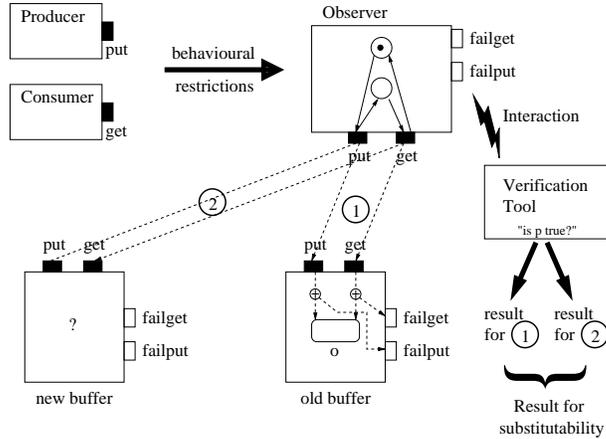


Figure 24: Substitutability of buffers

Please note that the *putfail* and *getfail* gates of the two buffers are connected to those of the observer, but we have chosen not to represent the connections on the picture for clarity reasons. Our validation formula is something like: “put @ implies there exists a path from the current state where we can fire put @ and sometime in the future fire get @”.

The result of the application of a verification tool on the observer composed with, successively, the old and new buffer must give the same result. Only then

can we state that the new buffer is a subtype of the old buffer with respect to our observer and our validation formula, and only then can we operate the substitution.

## 4 Conclusions and Future Work

We have presented in this work an original formal definition of subtyping and substitutability for a component-oriented formalism. The aim was to have a very general subtype relation, which we could specialize depending on our requirements, by either making it very strong or very weak, and making it depend from its context of use or not. We have proposed in this paper an example of such a relation where we force it to compare “strongly” behaviour of components of some sub-transition system of these components, and we have called this specialized subtype relation *exterior system* subtype relation. One application of this it to simplify verification in the sense that, instead of checking a set of properties of a component, we check this set of properties on the same component composed with an observer, thus reducing its transition system (its semantics) and the complexity of the verification.

We are now making a survey of all the existing subtype relations that can be brought in our framework (such as Liskov’s, for example). Work is also currently being conducted on the verification aspect. For instance, we are experimenting using PVS [5] to try to prove properties on components. We are also working on properties verification on the CO-OPN formalism (where the general problem first appeared to us), which basically comes down to properties verification on algebraic nets. We still need to add a subtype module to our tool [6], which would allow to define and check subtypes for the above formalism.

## References

- [1] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi; Object-oriented nets with algebraic specifications: the CO-OPN/2 formalism. In Lecture Notes in Computer Science. Springer-Verlag, LNCS 2001, pp. 70-127.
- [2] Didier Buchs, Sandro Costa, David Hurzeler; Component Based System Modelling for Easier Verification; In P. Ezhilchevan and A. Romanovski, editors, Concurrency in Dependable Computing, pp.61–86, Kluwer, 2002.
- [3] Barbara Liskov and Jeannette M. Wing; A behavioural notion of subtyping, ACM Transaction on Programming Languages and Systems, 16(6):1811–1841, November 1994.
- [4] M. Hennessy, R. Milner; Algebraic Laws for Nondeterminism and Concurrency. Journal of the ACM, 32(1):137-161, jan 1985.
- [5] Sam Owre, John M. Rushby, and Natarajan Shankar; PVS: A prototype verification system. In Deepak Kapur, editor, Automated Deduction - CADE-11 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, pp 748–752, Springer-Verlag, June 1992.
- [6] <http://lglwww.epfl.ch/Conform/CoopnTools>

- [7] N. Hameurlain, Behavioural Subtyping and Property Preservation for Active Objects; In Bart Jacobs, Arend Rensink, editors, Formal Methods for Open-Based Distributed Systems V, pp.95-110, Kluwer, 2002.
- [8] M. Hennessy, C. Stirling; The Power of the Future Perfect in Program Logics. Information and Control, 67:23-52, 1985
- [9] R. Milner; A Modal Characterisation of Observable Machine Behaviour. In Proc. CAAP81, Genoa, LNCS 185, p.25-34. Springer-Verlag, March 1981.
- [10] P. Schnoebelen, Semantique du parallélisme et logique temporelle, Application au langage FP2, Phd Thesis, Institut National Polytechnique de Grenoble, june 1990.