Assembling Components with Behavioural Contracts
Assemblage de Composants selon des Contrats Comportementaux

Cyril Carrez[1,2] **— Alessandro Fantechi**[3,4] **— Elie Najm**[2]

[1] *Department of Telematics, Norwegian University of Science and Technology*
O.S. Bragstads plass 2B, N-7491 Trondheim, Norway
cyril.carrez@item.ntnu.no

[2] *Ecole Nationale Supérieure des Télécommunications, Département INFRES*
46 rue Barrault, F-75013 Paris, France
elie.najm@enst.fr

[3] *Universitá di Firenze, Dipartimento di Sistemi e Informatica*
Via S. Marta 3, I-50139 Firenze - Italy
fantechi@dsi.unifi.it

[4] *ISTI - CNR; Via G. Moruzzi 1, I-56124 Pisa - Italy*

ABSTRACT. *Component based design is a new paradigm to build distributed systems and applications. The problem of compositional verification of such systems is however still open. We investigate methods and concepts for the provision of "sound" assemblies. We define a behavioural interface type language endowed with a (decidable) set of interface compatibilty and subtyping rules. We define an abstract, dynamic, multi-threaded, component model, encompassing both client/server and peer to peer communication patterns. Based on the notion of compliance of components to their interfaces, we define the concepts of "contract" and "contract satisfaction". This leads to sound assemblies of components, which possess intersting properties, such as "external deadlock freeness" and "message consumption".*

RÉSUMÉ. *La conception basée composants est une nouvelle méthode de construction d'applications et de systèmes distribués. La vérification compositionelle de ces systèmes reste cependant un problème. Nous étudions des méthodes et des concepts pour la construction d'assemblages "sains". Nous définissons un langage de type d'interfaces comportementales, doté d'un ensemble de règles (décidables) de compatibilité d'interface et de sous-typage. Nous définissons un modèle de composant abstrait, dynamique et multi-tâche, qui englobe les modèles client/serveur et point-à-point. Basé sur la notion de conformité du composant par rapport à son interface, nous définissons les concepts de "contrat" et "respect de contrat". Cela mène aux assemblages sains de composants qui possèdent des propriétés intéressantes, comme "l'absence d'interblocage externe" et "la consommation des messages".*

KEYWORDS: *Behavioural typing, verification, composition, components, peer-to-peer, client/server*

MOTS-CLÉS : *Typage comportemental, vérification, composition, composants,* peer-to-peer, *client/serveur*

1. Introduction

Behavioural type systems have been defined in recent years with the aim to be able to check the compatibility of communicating concurrent objects, not only regarding data exchanged, but also regarding the matching of their respective behaviour [Nie95, KPT99, NNS99]. This check finds a natural application in the verification of compatibility of components, as the recent advances in Software Engineering are towards *component-based design*: a software system is developed as a construction based on the use of components connected together either by custom-made glue code, or by resorting to a standard platform supporting composition and communication, such as CORBA or .NET. The compatibility of a component with its environment has to be guaranteed before it is deployed.

Formal verification techniques can therefore play a strategic role in the development of high quality software. In the spirit of the so-called *lightweight formal methods*, the software engineer who connects components is not bothered by a formal description of the software artifact he is building, but gets a guarantee about the absence of mismatches between components; this guarantee is provided by the underlying formally verified components and by the formal verification algorithms that check type compatibility. An even more demanding example is mobile code, where one needs the guarantee that a migrating component does not undermine the correctness of the components that it reaches. This check has to be performed at run-time, at the reception of the migrating component, and hence has to be performed very efficiently. Typing of mobile agents has already been addressed for example in [HR02], but we aim at a more abstract behaviour of the component which is sufficient to efficiently prove that desired properties of the global configuration of components are not endangered by the composition.

In this work we define a framework in which a component can exhibit several *interfaces* through which it communicates with other components. Each interface is associated a type, which is an abstraction of the behaviour of the component. Our type language (for interfaces) introduces modalities on the sequences of actions to be performed by interfaces. Using **must** and **may** prefixes, it allows the distinction between *required* messages and *possible* ones. The complexity of the interface typing language is kept deliberately low, in order to facilitate compatibility verification among interfaces. We do not give a specific language for components, but we rather give an abstract definition, which wants to be general enough to accomodate different languages: indeed, components are abstracted as a set of ports, by which they communicate, together with a set of internal threads of execution, of which we observe only the effects on the ports. Under given constraints on the use of ports inside components, it is shown that a configuration made up of communicating components satisfies well-typedness and liveness properties, if the components honour the contracts given them by their interfaces, and the communicating interfaces are compatible. This work extends our previous one [CFN03b, CFN03a] with subtyping and a dual relation, and how they relate to the compatibility relation; we also put our work into practice with a more complex example.

Our work is partly inspired by that of De Alfaro and Henzinger [dAH01], who associate interface automata to components and define compatibility rules between interfaces. Our approach, which belongs instead to the streamline of process algebraic type systems, brings in the picture also the compliance between components and interfaces: the interface is thought as a *contract* with the environment, that the component should honour. We also aim at limit-

ing as much as possible the complexity of the interface compatibility check, which can even be needed to be performed at run-time. The work on Modal Transition Systems by Larsen, Steffen and Weise [LSW95] has inspired our definition of modalities and the way interface compatibility is checked. The guarantee of the satisfaction of well-typedness and liveness properties has been dealt by Najm, Nimour and Stefani [NNS99], and we have inherited their approach in showing how the satisfaction of compatibility rules guarantees more general properties.

This paper is structured as follows: in Section 2 we show our interface language and the related compatibility and subtyping rules, whereas Section 3 deals with the reference component model. Interface language and component model are put together in the concept of *component honouring a contract* (Sect. 4), which leads to interesting properties that can be guaranteed by sound assemblies of components (Sect. 5). Our work is illustrated with the classic bank account example in Section 6.

2. Interface Types

In this section we describe the language used to define the interfaces. A typed component is a component whereby every initial port or created port has an associated type, and every reference to a port has a declared type. We adopt a behavioral type language [Nie95, KPT99, NNS99]. In this setting, the type of a port prescribes its possible states, and for each state, the actions that are allowed and/or required through that port, and its state after the performance of an action. The BNF table below defines the syntax of types. Among the salient features of this type language is the use of **may** and **must** modalities. We also present a subtype relation and a compatibility relation.

2.1. *Syntax of the Interface Language*

The interface language has the following syntax:

$$
\begin{array}{rcl}
type & ::= & server_name \;\textbf{=}\; mod\ receive_server \\
 & | & peer_name \;\textbf{=}\; (mod\ send \mid mod\ receive) \\
receive_server & ::= & \textbf{? *[}\ \sum_i M_i\textbf{;}\ I_i\ \textbf{]} \\
receive & ::= & \textbf{? [}\ \sum_i M_i\textbf{;}\ I_i\ \textbf{]} \\
send & ::= & \textbf{! [}\ \sum_i M_i\textbf{;}\ I_i\ \textbf{]} \\
I & ::= & \textbf{0} \mid peer_name \mid mod\ send \mid mod\ receive \\
mod & ::= & \textbf{may} \mid \textbf{must} \\
M & ::= & name[\ \textbf{(}\ \widetilde{args}\ \textbf{)}\ ] \\
args & ::= & basic_type \mid peer_name \mid server_name \\
basic_type & ::= & \textbf{boolean} \mid \textbf{integer} \mid \textbf{real} \mid \textbf{string}.
\end{array}
$$

The **!** and **?** keywords are the usual sending and receiving actions. The choice operator **+** allows to choose one message among the list[1], and the **;** is used to sequence behaviors. The ***** keyword allows specification of servers, and is discussed later in this section. **0** (zero) stands for the termination of the interface. The modalities **may** and **must** distinguish between permissions and obligations for the performance of the actions. Their meaning is (where the partner is the other communicating port):

may ? [ΣM_i**;** I_i **]** "the port does not impose any sending constraint on the partner, but if the partner sends any message M_i, then the port guarantees to be ready to receive it, and to execute later on action I_i".

must ? [ΣM_i**;** I_i **]** "the port does impose a sending constraint on the partner, and if the partner sends any message M_i, then the port guarantees to be ready to receive it, and to execute later on action I_i".

may ! [ΣM_i**;** I_i **]** "the port may send one of the messages M_i to its partner and execute later on action I_i; the partner must be ready to receive the message M_i".

must ! [ΣM_i**;** I_i **]** "the port guarantees to send one of the messages M_i to its partner and execute later on action I_i; the partner must be ready to receive the message M_i".

Messages contain arguments. Thus, basic data (such as integers, booleans…) as well as references to ports (be it *peer_name* or *server_name*), can be passed in messages. Our type language does not cater for structured data, but their addition is straightforward. Sending or receiving references implies some restrictions that are enforced on the behaviour of the involved components:

! m (I) "the port is sending to its partner a reference to a port whose behavior is described by the type I. Moreover, the first action of this referenced port must be **?** (reception)."

? m (I) "the port is receiving a reference to another port whose behavior is conform to the type I. Moreover, the first action of this referenced port is a **?**."

The constraint on the first action of the referenced port is inevitable: if the first action of I is **!**, then the referenced port can send this message to a third port, which will lead to incompatible behaviours between components.

Finally, the *****-construct allows specification of a server. This server spawns to answer a request, so it immediately reconfigures to honour other potential clients:

$I =$ *mod* **? *[** $m()$**;** I' **]** after the reception of m, a port whose behaviour is I' is created while the server is regenerated as I. The new port will interact with the sender of the request. The first action of I' must be **!** (send), so the client will be informed of the port created for its request.

Several client ports can be connected at the same time to the same server port. As for peer ports, they are connected only in pairs. This distinction allows client-server and peer-to-peer links.

1. We consider that all messages of this list are distinct.

As regards to server types, we remark that a type (peer or server) cannot become a type declared as server. For example, the following declarations are forbidden (type I becomes server type $Server$; type $Server$ becomes server type $Server$):

$$I \textbf{ = may ! [} M\textbf{; } Server \textbf{]} \qquad Server \textbf{ = may ? *[} M\textbf{; } Server \textbf{]}$$

Our type language has the effect of not only imposing constraints on the component, but also of imposing constraints on its partner (i.e. on the environment). The introduction of modalities leads to an underlying model which is a kind of *modal* $\mathcal{LTS}$, in which states can be either **may** or **must** [LSW95] . This has a strong impact on the type compatibility rules, which are discussed in Section 2.3

2.2. *Subtyping*

In this section, we define a subtype relation, which has the classic property of subtypes (meaning the subtype can replace the supertype). The main idea of T being a subtype of S, noted $T \preceq S$, is the following:

– If T and S are receiving, then:
  - T receives *at least* the messages S can receive;
  - the subtype relation is *contra-variant*;
  - if S has **may** modality, then T has **may** modality. Otherwise there is no restriction.

– If T and S are sending, then:
  - T sends *at most* the messages S can send;
  - the subtype relation is *co-variant*;
  - if S has **must** modality, then T has **must** modality. Otherwise there is no restriction.

For example, with a lightened syntax, we have:

$$\begin{array}{rcl}
\textbf{may ?}\ M_1 + M_2 + M_3 & \preceq & \textbf{may ?}\ M_1 + M_2 \\
(\textbf{may}|\textbf{must})\ \textbf{?}\ M_1 + M_2 + M_3 & \preceq & \textbf{must ?}\ M_1 + M_2 \\
(\textbf{may}|\textbf{must})\ \textbf{!}\ M_1 & \preceq & \textbf{may !}\ M_1 + M_2 \\
\textbf{must !}\ M_1 & \preceq & \textbf{must !}\ M_1 + M_2
\end{array}$$

We further explain our choice on the subtyping of modalities. When receiving, if the supertype is **may?**, it does not impose any constraint on its partner. Hence the subtype cannot impose any constraint either, which leads to the **may?** subtype. Similar reasoning explains other subtype relations.

The subtype relation is such that it also recursively applies to the type of the actions following reception and sending: if $I =$ **may ! [** M_1**;**I_1 **]** is a subtype of $J =$ **may ! [** M_1**;**J_1 **+** M_2**;**J_2 **]** then I_1 is a subtype of J_1.

The subtype relation is formally described by the rules in Tab. 1 and the subtype predicate $\preceq_m$ on modalities. Three cases in this predicate deserve a few lines.

$T_{\mathrm{mod}} \preceq_m S_{\mathrm{mod}}$

T_{mod} \ S_{mod}	**must?**	**may?**	**must!**	**may!**	**0**
must?	$\surd$				
may?	$\surd$	$\surd$		$\surd$	$\surd$
must!			$\surd$	$\surd$	
may!				$\surd$	
0				$\surd$	$\surd$

$$\text{ST-BOOL}\ \frac{}{\Gamma \vdash \textbf{boolean} \preceq \textbf{boolean}} \qquad \text{ST-ASSUMP}\ \frac{I \preceq J \in \Gamma}{\Gamma \vdash I \preceq J}$$

$$\text{ST-STRING}\ \frac{}{\Gamma \vdash \textbf{string} \preceq \textbf{string}} \qquad \text{ST-NAME1}\ \frac{\Gamma, \mathit{name} \preceq I \vdash \mathrm{replace}(\mathit{name}) \preceq I}{\Gamma \vdash \mathit{name} \preceq I}$$

$$\text{ST-INT}\ \frac{}{\Gamma \vdash \textbf{integer} \preceq \textbf{integer}} \qquad \text{ST-NAME2}\ \frac{\Gamma, I \preceq \mathit{name} \vdash I \preceq \mathrm{replace}(\mathit{name})}{\Gamma \vdash I \preceq \mathit{name}}$$

$$\text{ST-INT2}\ \frac{}{\Gamma \vdash \textbf{integer} \preceq \textbf{real}}$$

$$\text{ST-MAYRECV}\ \frac{}{\Gamma \vdash \textbf{may?}\ [\textbf{*}]\textbf{[}\ \Sigma_i M_i(\tilde{I}_i)\textbf{;}\ T_i\ \textbf{]} \preceq \textbf{0}}$$

$$\text{ST-REAL}\ \frac{}{\Gamma \vdash \textbf{real} \preceq \textbf{real}}$$

$$\text{ST-MAYSEND}\ \frac{}{\Gamma \vdash \textbf{0} \preceq \textbf{may !}\ \textbf{[}\ \Sigma_i M_i(\tilde{I}_i)\textbf{;}\ T_i\ \textbf{]}}$$

$$\text{ST-ZERO}\ \frac{}{\Gamma \vdash \textbf{0} \preceq \textbf{0}}$$

$$\text{ST-RECVSEND}\ \frac{}{\Gamma \vdash \textbf{may ?}\ [\textbf{*}]\textbf{[}\ \sum_{1\leqslant i\leqslant n} N_i(\tilde{J}_i)\textbf{;}\ T_i\ \textbf{]} \preceq \textbf{may !}\ \textbf{[}\ \sum_{1\leqslant i\leqslant m} M_i(\tilde{I}_i)\textbf{;}\ S_i\ \textbf{]}}$$

$$\text{ST-RECV}\ \frac{\forall i \in \{1..m\} : \Gamma \vdash \tilde{I}_i \preceq \tilde{J}_i\ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash \mathit{mod}_T\ \textbf{?}\ [\textbf{*}]\textbf{[}\ \sum_{1\leqslant i\leqslant n} M_i(\tilde{J}_i)\textbf{;}\ T_i\ \textbf{]} \preceq \mathit{mod}_S\ \textbf{?}\ \textbf{[}\ \sum_{1\leqslant i\leqslant m} M_i(\tilde{I}_i)\textbf{;}\ S_i\ \textbf{]}}\ \Box$$

$$\text{ST-RECV*}\ \frac{\forall i \in \{1..m\} : \Gamma \vdash \tilde{I}_i \preceq \tilde{J}_i\ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash \mathit{mod}_T\ \textbf{?}\ \textbf{*[}\ \sum_{1\leqslant i\leqslant n} M_i(\tilde{J}_i)\textbf{;}\ T_i\ \textbf{]} \preceq \mathit{mod}_S\ \textbf{?}\ \textbf{*[}\ \sum_{1\leqslant i\leqslant m} M_i(\tilde{I}_i)\textbf{;}\ S_i\ \textbf{]}}\ \Box$$

$$\text{ST-SEND}\ \frac{\forall i \in \{1..n\} : \Gamma \vdash \tilde{J}_i \preceq \tilde{I}_i\ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash \mathit{mod}_T\ \textbf{!}\ \textbf{[}\ \sum_{1\leqslant i\leqslant n} M_i(\tilde{J}_i)\textbf{;}\ T_i\ \textbf{]} \preceq \mathit{mod}_S\ \textbf{!}\ \textbf{[}\ \sum_{1\leqslant i\leqslant m} M_i(\tilde{I}_i)\textbf{;}\ S_i\ \textbf{]}}\ \Diamond$$

$$\Box \triangleq (\mathit{mod}_T\ \textbf{?} \preceq_m \mathit{mod}_S\ \textbf{?})\ \wedge\ n \geqslant m$$

$$\Diamond \triangleq (\mathit{mod}_T\ \textbf{!} \preceq_m \mathit{mod}_S\ \textbf{!})\ \wedge\ n \leqslant m$$

$$\tilde{I} \preceq \tilde{J} \triangleq \tilde{I} = (I_i)_{1..k}, \tilde{J} = (J_i)_{1..k}\ \wedge\ \forall i, I_i \preceq J_i$$

Table 1. *Subtyping rules* ([*****] means the *****-construction is optional)

may? $\preceq_m$ **0** and **0** $\preceq_m$ **may!** illustrate the semantics of **may** modality: **may?** can replace an inactive interface **0** (no message will be sent to the inactive interface **0**, hence no message will be sent to the **may?** interface). Also, an inactive interface **0** can be considered as an interface that can send messages (but with no obligation). Finally, because subtype relations are transitive, we have another relation: **may?** $\preceq_m$ **may!**. At first sight, this relation seems strange; however, by looking at the semantics of modalities, we have a supertype which *may* send, and a subtype **may?** which will never send messages, thus respecting the **may!** semantics. This relation is surprising especially because one is used to conceive sendings and receivings with the **must** modality.

Once the principles of the $\preceq_m$ relation are understood, subtying rules are quite straightforward. The first six ones (on the left column) deal with basic types and inactive interfaces. Rules ST-ASSUMP, ST-NAME1 and ST-NAME2 take into account recursive interfaces, and cases where interface is pointed out using its name ($\mathrm{replace}(\textit{name})$ replaces *name* with the definition of the corresponding type). Rules ST-MAYRECV and ST-MAYSEND concern subtyping staking type **0**. The rule ST-RECVSEND tackles the **may?** $\preceq_m$ **may!** relation: note the message lists between sub- and super-type are different. The three last rules give the classical definition of interface subtypes: the subtype can receive more, and send less than its supertype. Order of messages is not considered of importance (as all messages are distinct).

Concerning server interfaces, a subtype can be a server (while the supertype is not), but the opposite is forbidden, as shown with the rule ST-RECV*: this rule is identical to ST-RECV, but applies to server types (both sub- and supertypes). This restriction is due to the fact that server ports can be connected to several ports: if the supertype $S^\star$ is a server port, then this port can answer several requests, which will not be the case of a port whose subtype $T \preceq S^\star$ is not a server. On the contrary, $T^\star \preceq S$ does not raise this kind of problems.

Property 1
$\preceq$ is a partial preorder.

Proof. Truth table of $\preceq_m$ assures transitivity and reflexivity among modalities. The rest of the demonstration is made by structural induction. □

2.3. *Compatibility Rules*

In this section we define the symmetric predicate $Comp(I, J)$ as "I and J are compatible with each other". Compatibility between interfaces I and J is informally defined as follows:

$I =$ **must ?** m implies $J =$ **must !** m
$I =$ **may ?** m implies $J =$ **must !** m or $J =$ **may !** m or $J =$ **0** or $J =$ **may ?** m'
$I =$ **must !** m implies $J =$ **must ?** m or $J =$ **may ?** m
$I =$ **may !** m implies $J =$ **may ?** m
$I =$ **0** implies $J =$ **may ?** m or $J =$ **0**

The compatibility rules are actually defined using several elementary compatibility relations: compatibility between modalities, messages, and finally types.

We first define the compatibility between modalities, as the symmetric boolean relation $Comp_{\text{mod}}(mod_I\ [!|?], mod_J\ [!|?])$. Its truth table is reproduced hereafter:

J	I must?	may?	must!	may!	0
must?			$\surd$		
may?		$\surd$	$\surd$	$\surd$	$\surd$
must!	$\surd$	$\surd$			
may!		$\surd$			
0		$\surd$			$\surd$

We define also $Comp_{\text{msg}}$, a relation over message types. Two message types are compatible iff they have the same name and their arguments are pairwise related such that arguments of sent messages are subtypes of arguments of received messages. This is formally defined (with $M_{\mathbf{!}}$ the sent message, and $M_{\mathbf{?}}$ the received one):

$$Comp_{\text{msg}}(M_{\mathbf{!}}, M_{\mathbf{?}}) \triangleq Comp_{\text{msg}}(M_{\mathbf{!}}(I_1, \ldots, I_n), M_{\mathbf{?}}(J_1, \ldots, J_n))$$
$$\triangleq M_{\mathbf{!}} = M_{\mathbf{?}} \wedge \forall i, I_i \preceq J_i$$

We can then define the compatibility $Comp(I, J)$ between two interfaces as compatibility between modalities and messages, and transitions must lead to compatible interfaces. This is formally defined recursively as (with $\rho \in \{\mathbf{?}, \mathbf{!}\}$, and where [*] means that the *-construct may be present or not):

$$
\begin{array}{rcl}
Comp(I, J) & \triangleq & Comp(J, I) \\
Comp(\mathbf{0}, \mathbf{0}) & \triangleq & true \\
Comp(\mathbf{0}, mod_J\ \rho_J\ [\mathbf{*}]\mathbf{[}\ \Sigma_l\ M'_l\mathbf{;}\ J_l\ \mathbf{]}\ ) & \triangleq & Comp_{\text{mod}}(\mathbf{0}, mod_J\ \rho_J) \\
Comp(\mathbf{may?}\ [\mathbf{*}]\mathbf{[}\ \Sigma_l\ M_k\mathbf{;}\ I_k\ \mathbf{]}, & & \\
\mathbf{may?}\ [\mathbf{*}]\mathbf{[}\ \Sigma_l\ M'_l\mathbf{;}\ J_l\ \mathbf{]}) & \triangleq & true \\
Comp(mod_I\ \mathbf{!}\ \mathbf{[}\ \Sigma_k\ M_k\mathbf{;}\ I_k\ \mathbf{]}\ , & \triangleq & Comp_{\text{mod}}(mod_I\ \mathbf{!}, mod_J\ \mathbf{?}) \\
mod_J\ \mathbf{?}\ [\mathbf{*}]\mathbf{[}\ \Sigma_l\ M'_l\mathbf{;}\ J_l\ \mathbf{]}\ ) & & \wedge\ \big(\forall k, \exists l : Comp_{\text{msg}}(M_k, M'_l) \wedge Comp(I_k, J_l)\big)
\end{array}
$$

The recursive definition indicates that the compatibility of a pair of interfaces is a boolean function of a finite set of pairs of interfaces. This definition also closely resembles the definition of simulation or equivalence relations over finite state transition systems. Hence, the verification of compatibility always terminates, and can be performed with standard techniques in a quadratic complexity with the number of interfaces (intended as different states of the interfaces). Due to the abstraction used in the definition of interfaces, such number is small with respect to the complexity of the component behaviour. Moreover, the wide range of techniques introduced for the efficient verification of finite state systems can be studied in search of the ones that best fit this specific verification problem.

2.4. *Properties of the Subtypes*

In this section, we prove the classical property of subtypes, i.e. that a subtype can replace supertype without any change of the partner of the supertype:

Property 2
If $T \preceq S$ *then* $\forall I, Comp(I,S)$ *implies* $Comp(I,T)$

Lemma 1 (subtyping and modalities)
If $T_{\mathrm{mod}} \preceq_m S_{\mathrm{mod}}$ *then* $\forall I_{\mathrm{mod}}, Comp_{\mathrm{mod}}(I_{\mathrm{mod}}, S_{\mathrm{mod}})$ *implies* $Comp_{\mathrm{mod}}(I_{\mathrm{mod}}, T_{\mathrm{mod}})$.

Proof. Straightforward from truth tables of $\preceq_m$ and $Comp_{\mathrm{mod}}$ (by looking only at cases where T and S modalities are different, there are 5 cases). □

Proof (prop. 2).
By induction on the structure of the types. We omit the case where S is a server, but proof is identical. Given T, S and I such that $T \preceq S$ and $Comp(I,S)$. There is three cases:

1) S is receiving.

Let's write $S = mod_S$ **? [** $\sum_{1 \leqslant l \leqslant m} M_l^S(\tilde{J}_l^S)$**;** S_l **]**. Receiving supertypes are concerned only by rules ST-RECV, ST-RECV*, which give $T = mod_T$ **?** [*]**[** $\sum_{1 \leqslant l \leqslant n} M_l^T(\tilde{J}_l^T)$**;** T_l **]** with: mod_T **?** $\preceq_m mod_S$ **?**, $n \geqslant m$ and $\forall l \in \{1..m\}: \tilde{J}_l^S \preceq \tilde{J}_l^T,\ T_l \preceq S_l,\ M_l^T = M_l^S$.

Compatibility between I and S has the following cases:

$I = \mathbf{0}$ or $I =$ **may?** [*]**[** $\sum_k M_k^I(\tilde{J}_k^I)$**;** I_k **]**.
Then we have $Comp(I,T) = Comp_{\mathrm{mod}}(mod_I, mod_T$ **?**$) =$ true, by lemma 1.

$I = mod_I$ **! [** $\sum_k M_k^I(\tilde{J}_k^I)$**;** I_k **]**.
Then $Comp(I,S)$ is written:
$Comp_{\mathrm{mod}}(mod_I$ **!**$, mod_S$ **?**$)\ \wedge\ \forall k, \exists l : Comp_{\mathrm{msg}}(M_k^I, M_l^S) \wedge Comp(I_k, S_l)$.
By lemma 1, the left term of the conjunction implies $Comp(mod_I$**!**$, mod_T$**?**$)$. As for the right term, we have by definition $Comp_{\mathrm{msg}}(M_k^I, M_l^S) \triangleq M_k^I = M_l^S \wedge \tilde{J}_k^I \preceq \tilde{J}_l^S \triangleq M_k^I = M_l^T \wedge \tilde{J}_k^I \preceq \tilde{J}_l^T$ by definition of $T \preceq S$ and transitivity of $\preceq$. Moreover, as $Comp(I_k, S_l)$ is true, we have, by induction, $T_l \preceq S_l$, and by lemma 1: $Comp(I_k, T_l)$ is true, which leads to the conclusion.

2) S is sending (the reasoning is similar to the previous case).

Cases where T is not sending are trivial: S has then **may!** modality, thus I has **may?** modality which is compatible with **0** and **may?**, hence with T.

Suppose T is sending, and let's write $S = mod_S$ **! [**$\sum_{1 \leqslant l \leqslant m} M_l^S(\tilde{J}_l^S)$**;**$S_l$**]** and $T = mod_T$ **! [**$\sum_{1 \leqslant l \leqslant n} M_l^T(\tilde{J}_l^T)$**;**$T_l$**]**, with $n \leqslant m$, mod_T **!** $\preceq_m mod_S$ **!** and
$\forall i \in \{1..n\}: \tilde{J}_i^T \preceq \tilde{J}_i^S \wedge T_i \preceq S_i \wedge M_i^T = M_i^S$. I has to be receiving:
$I = mod_I$ **? [** $\sum_k M_k^I(\tilde{J}_k^I)$**;** I_k **]**. Compatibility between I and S give:
$Comp_{\mathrm{mod}}(mod_S$ **!**$, mod_I$ **?**$) \wedge \forall k \in \{1..m\}, \exists l : Comp_{\mathrm{msg}}(M_k^S, M_l^I) \wedge Comp(S_k, I_l)$.
Particularly: $\forall k \leqslant n \leqslant m, \exists l : Comp_{\mathrm{msg}}(M_k^S, M_l^I) \wedge Comp(S_k, I_l)$. As $M_k^T = M_k^S$, $\tilde{J}_k^S \preceq \tilde{J}_k^T$ and $T_k \preceq S_k$, we are sure that, for any message from T, there exists a message from I satisfying $M_k^T = M_l^I \wedge \tilde{J}_k^T \preceq \tilde{J}_l^I \wedge Comp(T_k, I_l)$. Finally lemma 1 allows to conclude.

3) S has no action ($S = \mathbf{0}$).

Then T has **may?** modality, and I has either **may?** modality, or is inactive (**0**). $Comp(I,T)$ is straightforward. □

2.5. *Dual Interfaces*

We define a duality relation, noted $.^{\mathcal{D}}$, for peer interfaces, such that $I^{\mathcal{D}}$ is unique and compatible with I. The dual of I is simply calculated by exchanging sendings and receivings:

$$\mathbf{0}^{\mathcal{D}} \triangleq \mathbf{0}$$

$$\big(mod\ \mathbf{!}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ I_i\ \mathbf{]}\big)^{\mathcal{D}} \triangleq mod\ \mathbf{?}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ {I_i}^{\mathcal{D}}\ \mathbf{]}$$

$$\big(mod\ \mathbf{?}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ I_i\ \mathbf{]}\big)^{\mathcal{D}} \triangleq mod\ \mathbf{!}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ {I_i}^{\mathcal{D}}\ \mathbf{]}$$

The dual has the following properties (where the first three ones are obvious):

Property 3 (dual)

3.1) $I^{\mathcal{D}}$ *is unique;*

3.2) $(I^{\mathcal{D}})^{\mathcal{D}} = I$*;*

3.3) $Comp(I^{\mathcal{D}}, I)$*;*

3.4) $Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$*: the greater supertype compatible with* J *is* $J^{\mathcal{D}}$*;*

3.5) $I \preceq J \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$

Proof (prop. 3.4).
1.$\Rightarrow$: Proof is similar to the one of property 2. Sketch is the following.
Suppose I is sending. We have:
$I = mod_I\mathbf{![}\Sigma_k M_k^I(\tilde{I}'_k)\mathbf{;}I_k\mathbf{]}$ $\quad J = mod_J\mathbf{?[}\Sigma_l M_l^J(\tilde{J}'_l)\mathbf{;}J_l\mathbf{]}$ $\quad J^{\mathcal{D}} = mod_J\mathbf{![}\Sigma_l M_l^J(\tilde{J}'_l)\mathbf{;}{J_l}^{\mathcal{D}}\mathbf{]}$
The definition of compatibility give:
$Comp_{\text{mod}}(mod_I\mathbf{!}, mod_J\mathbf{?})$ and $\forall k, \exists l : M_l^I = M_k^J \wedge \tilde{I}'_k \preceq \tilde{J}'_l \wedge Comp(I_k, J_l)$
From which we deduce $mod_I\mathbf{!} \preceq mod_J\mathbf{!}$ and, with a rearrangement of messages:
$mod_I\mathbf{![}\Sigma_k M_k^I(\tilde{I}'_k)\mathbf{;}\ I_k\ \mathbf{]} \preceq mod_J\mathbf{![}\Sigma_l M_l^J(\tilde{J}'_l)\mathbf{;}\ J_l\ \mathbf{]}$ with $\tilde{I}'_k \preceq \tilde{J}'_l \wedge Comp(I_k, J_l)$
With the induction hypothesis, we have $I_k \preceq {J_l}^{\mathcal{D}}$. We conclude $I \preceq J^{\mathcal{D}}$.

Case $I = \mathbf{0}$ is similar: J is either $\mathbf{0}$, or **may ? [** . . . **]**, the dual of which is supertype of $\mathbf{0}$. Case I receiving is dealt the same as I sending: case J sending is identical. When J has **may?** modality, then I has the same modality, and the subtype relation we are looking for is **may?** $\preceq$ **may!**, which is true. Case $J = \mathbf{0}$ and $I =$ **may?[** . . . **]** leads to $I \preceq J^{\mathcal{D}}$.

2. $\Leftarrow$:
We have $Comp(J^{\mathcal{D}}, J)$ (prop. 3.3), from which we deduce $Comp(I, J)$ (prop. 2). □

Proof (prop. 3.5).
$I \preceq J \Leftrightarrow Comp(I, J^{\mathcal{D}}) \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$ □

2.5.1. *Duality and Server Types*

Our dual function can be extended to server types as follows (where the dual loses the *):

$$\big(mod\ \mathbf{?*[}\Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ I_i\ \mathbf{]}\big)^{\mathcal{D}} \triangleq mod\ \mathbf{!}\ \mathbf{[}\Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ {I_i}^{\mathcal{D}}\ \mathbf{]}$$

The dual is unique, but we no longer have, for a server I: $(I^{\mathcal{D}})^{\mathcal{D}} = I$. However, other properties hold, especially $Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$.

3. Component Model

3.1. *Informal Presentation*

Our computational model describes a system as a configuration of communicating components. Each component owns a set of ports, and communication is by asynchronous message passing between ports. Sending from a port can only occur if this port is bound to another "partner" port; then, any message sent from it is routed to this partner port. An unbound port can only perform receptions (this is the typical case for server ports). We consider dynamic configurations: a component may create new ports and may also dynamically bind a partner reference to any of its owned ports. In our setting, both peer to peer and client/server communications can be modelled: when two ports are mutually bound, they are peers; when the binding is asymmetrical, the bound port is a client and the unbound port is its server. We constrain peer references/ports to be private [NNS99], meaning a peer port is known only of its partner. Figure 1 shows a configuration made of three components. Note how port c (in C_1) is asymmetrically bound to s (in C_2; flag * indicates that reference s is a server), and the peer to peer binding between ports x and y (in C_2, C_3), and between u and v (both in C_1).

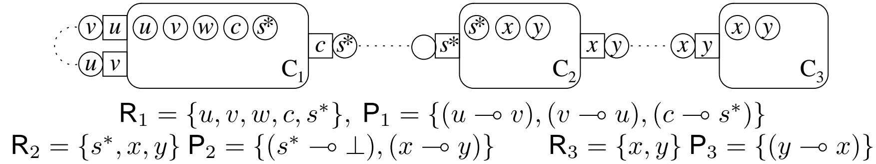


Figure 1. *An example of a configuration*

Components are also multi-threaded. We consider here an abstract thread model, focusing only on external, port based, manifestations of threads. Thus, an active thread is a chain made of a head port (the active port), and a tail (the ordered sequence of suspended ports). The thread chain may dynamically grow or decrease: this happens when the head port is suspended and the activity is given to another port, and when the head port is removed from the chain (because it terminated or became idle) and the port next to the head becomes active.

Since in this paper we focus on the interface typing issues, we do not provide a fully-fledged syntax for components. Rather, we define an abstract behavioural model of components in terms of their observable transitions and their multi-threaded, port-located, activities. The abstract model defined in this section is general and independent of any concrete behavioural notation for components.

3.2. *Notations for Components*

A component is made of a state, a set of ports, a set of references, and a set of threads.

The set of references is noted R, and is ranged over by u, v, w, c, s^*. The *-mark points out a reference to a server port. Classical set notation is used for the operations on R; however, we use the unorthodox $R \cup u$ notation for the insertion of element u into R, without duplicates.

The set of ports is noted P. It is a set of mappings from ports to partner references. Elements of P are noted $(\cdot \multimap \cdot)$. If a port u is bound to a partner v, then $(u \multimap v) \in \mathsf{P}$. If u has no partner, then $(u \multimap \bot) \in \mathsf{P}$. We also write $u \in \mathsf{P}$ if u is a local port belonging to P.

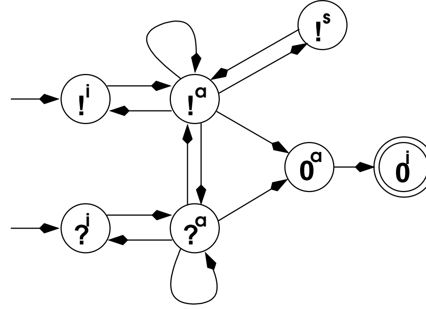
The following operators will be useful for the manipulation of port mappings:

$\mathsf{P}[u \multimap \bot]$	port u is added to P.
$\mathsf{P}[u \multimap v]$	bind the partner v to port u. Overrides the previous partner.
$\mathsf{P}\backslash u$	remove the port u from P.

The set of threads, T, reflects the state of the ports of the component and the dependencies between them. A port u depends (or suspends) on a port v when an action from u was not terminated, and this action needs some interaction from v (with v's partner). For example, in a proxy component, if u receives a request from a client, it will suspend to v which will forward the request to the server and recover the answer; then u will become active again and answer back to its client. The status of a port is abstracted from its activity (activated, suspended or idle) and from its state (sending, receiving or no action). We formally denote the activity state of a port u as $u\rho^\sigma$, with activity ρ and state σ such that:

$$\rho = \begin{cases} \mathbf{!} & u \text{ is in a sending state} \\ \mathbf{?} & u \text{ is in a receiving state} \\ \mathbf{0} & u \text{ has no action} \end{cases}$$

$$\sigma = \begin{cases} \mathsf{a} & u \text{ is active} \\ \mathsf{s} & u \text{ is suspended (by a port)} \\ \mathsf{i} & u \text{ is idle} \end{cases}$$



The state diagram shows the possible evolutions of the activity of a port. For example, $u\mathbf{!}^{\mathsf{a}}$ shows that the port u is active, and its next action is sending. This port can either send its message, be suspended by another port, or become idle. Some restrictions apply:

– when a port is created, it is in an idle state;

– $\mathbf{0}$ is terminal. $u\mathbf{0}^{\mathsf{a}}$ can only give back the thread of control, become $u\mathbf{0}^{\mathsf{i}}$ and vanish;

– $u\mathbf{!}^{\mathsf{a}}$ is an active port waiting to send a message. It can either send the message and change its state, suspend to another port, or become idle;

– $u\mathbf{?}^{\mathsf{a}}$ is an active port waiting for an incoming message. It can either receive the message and change its state, or become idle[2];

– only the sending ports can be suspended: combination $u\mathbf{?}^{\mathsf{s}}$ is forbidden.

We let x, y range over port activity states. We use the notation $x \rightarrowtail y$ which denotes x is suspended by y; this means that the activation of x is pending until y terminates (y has no action) or passivates (y becomes idle).

$\mathsf{T} = t_1 | \cdots | t_n$ is a set of parallel threads where a thread t is a sequence $x_1 \rightarrowtail x_2 \rightarrowtail \cdots \rightarrowtail x_n$. We note $|t|$ the length of thread t. This sequence has the following constraints:

2. However, a receiving port can become idle only if it will no longer receive messages (see Section 4).

$x_i = u_i\mathbf{!}^{\mathsf{s}}$ iff $i < |t|$ (all the ports but the last one are suspended)
$n = |t| > 1 \Rightarrow x_n = u_n\rho_n^{\mathsf{a}}$ (a sequence of more than one port ends with an active port)

In this paper, our model does not take into account dependencies between threads, for the sake of space. However, this extension is fully developed in [Car03].

The following operations on T are defined. We add a pool of idle ports: in the rules, ports are created and removed from this pool. Each port occurs only once in T.

$\mathsf{T} \mid u\rho^{\mathsf{i}}$	add a port in the idle ports pool.
$\mathsf{T}\backslash u$	remove u from T and activate the port that was possibly suspended by u. This operation is defined only if u is the head of some thread $t \in \mathsf{T}$.
$\mathsf{T}[u \rightarrowtail v]$	suspends port u to port v. This operation is defined only if u is the head of some thread $t_1 \in \mathsf{T}$ and v is in a singleton thread $t_2 = v\rho^{\mathsf{i}} \in \mathsf{T}$. It changes the state of u to suspended, adds the new head $v\rho^{\mathsf{a}}$ to t_1, and removes t_2. Note that a port can be head of only one thread at a time.
$\mathsf{T}[u\rho'/u\rho]$	modifies the state of a port in P: only ρ changes to ρ'.
$\mathsf{T}[u\rho^{\sigma\rightarrow\sigma'}]$	changes the activity of a port from σ to σ'.
$\mathsf{T}(u)$	returns ρ^σ if $u\rho^\sigma \in \mathsf{T}$.

3.3. *Communication Medium*

As indicated in the introduction, communication between components is by asynchronous message passing. Thus, a message is first deposited by its sender into a communication medium and, in a later stage, removed from this medium by its receiver. The delivery discipline that we adopt is first in first out. We define Com as a communication abstraction containing a collection of fifo queues, one for each port in the component: messages are written to and read from Com. We define the following notation on Com:

$Com[\triangleleft u]$	inserts a new queue for port u.
$Com.u$	the queue for port u. It is an ordered set of messages «$v : M(\tilde{w})$» where v is the reference of the sending port, M is the name of the message, and $\tilde{w}$ its arguments. If u is not defined, $Com.u = \bot$.
$Com\backslash u$	the queue u is removed.
$Com[u\triangleright]$	remove from the queue associated with u the next message.
$Com[u \triangleleft v : M(\tilde{w})]$	put message «$v : M(\tilde{w})$» in the queue associated with u.
$Com.u_{\triangleright}$	yields the next message (in queue u) to be treated.

3.4. *Component Semantics*

A component is defined by its state and its set of ports, references and threads:

$B(\mathsf{P}, \mathsf{R}, \mathsf{T})$ where	B is the state of the component; P, R, T are the ports, references and threads as defined previously.

The rules in Tab. 3 describe the semantics for the components, showing the transitions a component may perform in a given communication abstraction. A transition may change the state of the component itself and/or that of the communication abstraction. The actions of the

component deal essentially with handling of ports (creation/removal), references (bindings, removals), threads (activate/suspend) and messages (send/receive).

C-CREAT allows creation of a port in the component. The rule adds an unbounded and idle port, a reference to this port (so it can possibly be sent), and a corresponding queue in Com.
C-REMVPORT removes a port. It occurs only if the port is idle and its queue is empty.
C-REMVREF removes a reference from R. This reference must neither be a port ($u \notin \mathsf{P}$), nor bounded to a port ($u \notin CoDom(\mathsf{P})$).
C-BIND deals with binding of a partner. It is allowed only if the port has no partner ($(u \multimap \bot)$), if the port is sending, and not suspended ($\mathsf{T}(u) = \mathbf{!}^{\mathsf{a,i}}$)[3]. If the partner is a peer reference, it is bound to only one port at a time ($\mathrm{peer}(v) \Rightarrow v \notin CoDom(\mathsf{P})$).
C-ACTV allows the port u to suspend on a port v. u must be active, in a sending state ($\mathsf{T}(u) = \mathbf{!}^{\mathsf{a}}$), and v idle, in a sending state ($\mathsf{T}(v) = \mathbf{!}^{\mathsf{i}}$).
C-ACTV2 activates a port (from the pool of idle ports) with its own thread.
C-DEACT deactivates a port; this port has to be active.
C-SEND deals with sendings. The port must be active, in a sending state, bound to a reference ($(u \multimap u')$). Sent peer references must not be bound to a port ($\mathrm{peer}(\tilde{v}) \cap CoDom(\mathsf{P}) = \varnothing$), and must be removed from R ($\mathsf{R}' = \mathsf{R} - \mathrm{peer}(\tilde{v} \cup \{u\})$); as the reference to the port is also sent, it is removed from R if it is a peer. This ensures that peer references are known from only one component. After the sending, the port changes to the next action ($\mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{!}]$).
C-RECV deals with receivings. The sender becomes the new partner ($\mathsf{P}' = \mathsf{P}[u \multimap u'']$, only if u is a peer port), and the old one is removed ($\{u' | (u \multimap u')\}$ removed from R); this allows the partner to delegate its behaviour to another port. Received references and the new partner are stored in R. After the reception, the port changes to the next action ($\mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{?}]$).

3.5. *Configuration of Components*

When we take into account a configuration made up of several components, we consider the communication medium Com as shared among the components. This way, queues are shared and components can communicate through them, asynchronously. We give in Tab. 2 the communication rule for a configuration with two components, illustrating the sharing of Com. Extension to configurations with more components is straightforward.

$$\text{CPAR}\ \frac{B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1), Com \xrightarrow{\alpha} B_1'(\mathsf{P}_1', \mathsf{R}_1', \mathsf{T}_1'), Com'}{B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1) \mid B_2(\mathsf{P}_2, \mathsf{R}_2, \mathsf{T}_2), Com \xrightarrow{\alpha} B_1'(\mathsf{P}_1', \mathsf{R}_1', \mathsf{T}_1') \mid B_2(\mathsf{P}_2, \mathsf{R}_2, \mathsf{T}_2), Com'}$$

Table 2. *Rules for configurations of components*

4. Contract Satisfaction

The interface language presented in Section 2 imposes constraints on the remote interface, which will imply constraints also on the components. In this section, we present typing relation between components and the interface language, so the component will respect a

3. Binding a reference to a receiving port is useless as it will be erased during the reception (cf. rule C-RECV) and a binding when the port is suspended can be differed until the port is active.

$$\text{C-CREAT}\ \frac{\begin{array}{l}\mathsf{P}' = \mathsf{P}[u \multimap \bot] \qquad \mathsf{R}' = \mathsf{R} \cup u \qquad \mathsf{T}' = \mathsf{T} \mid u\rho^{\mathsf{i}} \\ Com' = Com[\triangleleft u]\end{array}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{creat}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ u \notin \mathsf{P}\ \wedge\ Com.u = \bot$$

$$\text{C-REMVPORT}\ \frac{\begin{array}{l}\mathsf{P}' = \mathsf{P}\backslash u \qquad \mathsf{R}' = \mathsf{R}\backslash u \qquad \mathsf{T}' = \mathsf{T}\backslash u \\ Com' = Com\backslash u\end{array}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ \mathsf{T}(u) = \rho^{\mathsf{i}}\ \wedge\ Com.u = \varnothing$$

$$\text{C-REMVREF}\ \frac{\mathsf{R}' = \mathsf{R}\backslash u}{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}\ u \notin \mathsf{P} \cup CoDom(\mathsf{P})$$

$$\text{C-BIND}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{bind}(u\multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T}), Com}\ \square$$

$$\text{C-ACTV}\ \frac{\mathsf{T}' = \mathsf{T}[u \rightarrowtail v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{actv}(u\rightarrowtail v)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \mathbf{!}^{\mathsf{a}}\ \wedge\ \mathsf{T}(v) = \mathbf{!}^{\mathsf{i}}$$

$$\text{C-ACTV2}\ \frac{\mathsf{T}' = \mathsf{T}[u\rho^{\mathsf{i}\to\mathsf{a}}]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{actv}(u)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \rho^{\mathsf{i}}$$

$$\text{C-DEACT}\ \frac{\mathsf{T}' = (\mathsf{T}\backslash u) \mid u\rho^{\mathsf{i}}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{deact}(u)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \rho^{\mathsf{a}}$$

$$\text{C-SEND}\ \frac{\mathsf{R}' = \mathsf{R} - \text{peer}(\tilde{v} \cup \{u\}) \qquad \mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{!}] \qquad Com' = Com[u' \triangleleft u : M(\tilde{v})]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u'\mathbf{!}M(\tilde{v})} B'(\mathsf{P},\mathsf{R}',\mathsf{T}'), Com'}\ \triangle$$

$$\text{C-RECV}\ \frac{\begin{array}{ll}\mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{?}] & \mathsf{R}' = \mathsf{R} \cup \{\text{refs}(\tilde{v}), u''\} - \{u' \mid (u \multimap u') \wedge \text{peer}(u')\} \\ Com' = Com[u\triangleright] & \mathsf{P}' = \mathsf{P}[u \multimap u''] \text{ if peer}(u)\end{array}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u''\mathbf{?}M(\tilde{v})} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ \triangledown$$

$$\square \triangleq (u \multimap \bot)\ \wedge\ \mathsf{T}(u) = \mathbf{!}^{\mathsf{a,i}}\ \wedge\ v \in \mathsf{R}\ \wedge\ \big(\text{peer}(v) \Rightarrow v \notin CoDom(\mathsf{P})\big)$$

$$\triangle \triangleq \mathsf{T}(u) = \mathbf{!}^{\mathsf{a}}\ \wedge\ (u \multimap u')\ \wedge\ u' \in Com\ \wedge\ \text{refs}(\tilde{v}) \subseteq \mathsf{R}\ \wedge\ \text{peer}(\tilde{v}) \cap CoDom(\mathsf{P}) = \varnothing$$

$$\triangledown \triangleq \mathsf{T}(u) = \mathbf{?}^{\mathsf{a}}\ \wedge\ Com.u_{\triangleright} = u'' : M(\tilde{v})$$

Table 3. *Rules for component semantics*

contract described by this language. The definitions of Section 3 are extended with the notion of contract. This notion is based on observers: the contract observes the actions of the component; if the action respects the contract, component and contract will evolve together.

A component has a set of contracts, one for each port. We use the notation:

$u\!:\!T, u\!<:\!T$ u has the behaviour described by the type T ($u\!:\!T$) or a subtype of T ($u\!<:\!T$)

$(B, \tilde{C})$ contract $\tilde{C}$ is associated to B. $\tilde{C}$ is a set of $(u : T)$, such that each reference (ports and partners) has an associated type. Addition of a reference is denoted $\tilde{C} \Leftarrow u\!:\!T$ (if $u\!:\!T'$ is already in $\tilde{C}$, there is no addition), and removal $\tilde{C} \backslash u$. Modification[4] is denoted $\tilde{C}[u\!:\!T'/T]$ when the type of u changes from T to T', and $\tilde{C}[u'\!:\!T'/u\!:\!T]$ when the reference $u\!:\!T$ is replaced by another reference $u' : T'$. We also write $u\!:\!T$ for $u\!:\!T \in \tilde{C}$, when there is no ambiguity (especially in the rules).

4.1. *Rules for Contract Satisfaction: Valid Transitions*

The rules (Tab. 4) are based on the ones of Sect. 3, whereby Com is abstracted from the state structure. Contracts are known locally: there is no global environment gathering them.

The predicate $\mathrm{Must}(\mathsf{T})$ states that any port u of the thread T which is typed **must!** is not suspended by a port v which is typed **may?**. This predicate is to be verified each time a port can change its state to a reception, meaning after a sending or a receiving. There is no need to verify this predicate when a port suspends to another one, because component semantics forbids the latter port to be receiving.

CREAT allows a port to be created. It adds the type of the port to $\tilde{C}$.
REMVPORT concerns removal of a port: its type is removed from $\tilde{C}$. No condition relates to the rule, but an active port in reception which can possibly receive a message cannot be removed. Indeed, rule DEACT prohibits an active port in reception to become idle, except in an exceptional case, where the corresponding port will no longer receive messages.
REMVREF removes a reference from R. It is allowed only if the reference is not typed **must?**: otherwise the component will have to interact with this reference, or send it.
BIND is very important, as it concerns binding of a reference to a port. Port and partner must have compatible types. This enables dynamic links that are valid.
DEACT authorises deactivation in the following cases: u is not in a receiving state («$u\!:\!T \not\equiv mod$ **?** $[^*]M_\Sigma$»), or the type of partner of u is a subtype of **0** (then u has the type **0** or **may?**). Regarding as to the second point, the port v will never send messages: corollary 2 in section 5.3 ensures that the real type of port v is a subtype of the type of the *reference* v, hence the *port* v is a subtype of **0**, and is not in a sending state[5].
SEND verifies the sendings. When u sends message M_k, then its type and the one of its partner u' will evolve accordingly. Sent references must be of the right type, and, for the peer references, removed from $\tilde{C}$ («$\mathrm{peer}(\tilde{v}_k \cap \overline{\mathsf{P}})$» gives peer references that are not local ports).
RECV verifies receptions of messages, when the port is bound to a partner (the type of this partner is in $\tilde{C}$: $u' : T' \in \tilde{C}$). Once reception is made, the type of the port and the one of its

---

4. Concerning servers: the contract keeps the initial type of the server. If $(B, \tilde{C})$ uses reference $g^* : T$, once the first message is sent, $\tilde{C}$ will contain both $g^* : T$ and $g : T'$, with T' the next type of the server.
5. The demonstration of the corollary given in section 5.3 is done by induction on the rules; the reasoning we gave may appear circular, but the corollary and this proof can be shown without rule DEACT.

For the sake of readability, we abbreviate:
$M_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}_k);\, T_k], M'_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}'_k);\, T'_k] \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{CREAT}\ \frac{u:T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{creat}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{creat}(u)} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u:T)}$$

$$\text{REMVPORT}\ \frac{u:T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{remvport}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}$$

$$\text{REMVREF}\ \frac{u:T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{remvref}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}\ (T \not\equiv \textbf{must?}\ M_\Sigma)$$

$$\text{BIND}\ \frac{u:T \qquad v:S \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{bind}(u\multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{bind}(u\multimap v)} (B'(\mathsf{P}',\mathsf{R},\mathsf{T}),\tilde{C})}\ \mathit{Comp}(T,S)$$

$$\text{DEACT}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{deact}(u)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{deact}(u)} (B'(\mathsf{P},\mathsf{R},\mathsf{T}'),\tilde{C})}\ \left(\begin{array}{c} u:T \not\equiv \mathit{mod}\ \textbf{?}\ [\textbf{*}]M_\Sigma \\ \vee\big((u \multimap v)\ \wedge\ v <: \mathbf{0}\big)\end{array}\right)$$

$$\text{SEND}\ \frac{u:T \equiv \mathit{mod}\ \textbf{!}\ M_\Sigma \qquad u':T' \equiv \mathit{mod}'\ \textbf{?}\ [\textbf{*}]M'_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\textbf{!}m_k} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u'\textbf{!}m_k} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{C}[u:T_k/T,u':T'_k/T']\backslash\{\text{peer}(\tilde{v}_k \cap \overline{\mathsf{P}})\}}\ \blacktriangleleft$$

$$\text{RECV}\ \frac{\begin{array}{l} u:T \equiv \mathit{mod}\ \textbf{?}\ M_\Sigma \\ u':T' \equiv \mathit{mod}'\ \textbf{!}\ M'_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u''\textbf{?}m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')\end{array}}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u''\textbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}[u:T_k/T,u'':T'_k/u':T'] \Leftarrow \tilde{v}:\tilde{U}'_k)}\ \blacktriangle \wedge (u \multimap u') \in \mathsf{P}$$

$$\text{RECV-UN}\ \frac{u:T \equiv \mathit{mod}\ \textbf{?}\ M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\textbf{?}m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u'\textbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}[u:T_k/T] \Leftarrow u':{T_k}^{\mathcal{D}},\tilde{v}:\tilde{U}_k)}\ \blacktriangle \wedge (u \multimap \bot) \in \mathsf{P}$$

$$\text{RECV*}\ \frac{u:T \equiv \mathit{mod}\ \textbf{?*}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:w\textbf{?}m_k,\text{creat}(u'),\text{bind}(u'\multimap w),\text{actv}(u')} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u'/u^\star:w\textbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u':T_k,w:{T_k}^{\mathcal{D}},\tilde{v}:\tilde{U}_k)}\ \blacktriangle$$

$$\text{OTHER}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\alpha} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\alpha} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C})}\ \alpha \in \{\text{actv}\}$$

$$\blacktriangleleft \triangleq \tilde{v}_k <: \tilde{U}_k \wedge \text{Must}(\mathsf{T}') \qquad \blacktriangle \triangleq \mathit{len}(\tilde{v}) = \mathit{len}(\tilde{U}_k) \wedge \text{Must}(\mathsf{T}')$$

$$\text{Must}(\mathsf{T}') \triangleq \forall u \in \mathsf{T}', (u:\textbf{must!}M_\Sigma) \Rightarrow \forall v, u \rightarrowtail^* v : \neg(v:\textbf{may?}M_\Sigma)$$

Table 4. *Rules for contract satisfaction (valid transition)*

partner evolve accordingly. Concerning the partner, modification in $\tilde{C}$ associates the type of the old partner (u', whose type T' becomes T'_k after reception) to the new partner (u'' which is then typed T'_k). Corallary 2 ensures that the true type of u'' is a subtype of T'_k.
RECV-UN deals with receptions when the port is not bound to a partner. Thus the type of the partner is unknown. We choose to associate to this partner the greater possible supertype, which is the dual type of the port receiving the message (prop. 3.4). Received references are stored with type $\tilde{U}_k$, which is necessarily a supertype of the real type of the references.
RECV* checks the correct operations of a server: the component must create a port at once, which will answer the request (sequence «$u : w\mathbf{?}m_k, \mathrm{creat}(u'), \mathrm{bind}(u' \multimap w), \mathrm{actv}(u')$»). Just like rule RECV-UN, the partner is given the dual type of that of the server port.
OTHER gives all other valid transitions, where the set $\tilde{C}$ is not modified.

We do not check the type of received arguments, because the message was sent according to the type of the sender; as the sender has to be compatible with the receiver, we are sure the arguments are well-typed. We aim also at an optimistic verification, where the environment has to respect the interface semantics: received messages not allowed by the type are not taken into account. Interactions between ports of the same component is also a bit tricky. Given u and v such ports, the component can behave such that u sends messages to v, knowing v is in the same component. Consequences of such a behaviour enforces the environment not to interact with port v. Just like the environment must respect the interface type it simulates, it must also respect privacy of peer references (i.e. it does not know reference v in this case).

4.2. *Rules for Contract Satisfaction: Invalid Transitions*

Rules of Tab. 5 give the error cases, where the component does not respect its contract.

REMVREF-ERR raises an error when a reference typed **must?** is removed from R.
BIND-ERR states that port and reference cannot be bound if their types are not compatible.
DEACT-ERR forbids a port in a receiving state ($u : mod\ \mathbf{?} \ldots$) to become idle if it has no partner, or if this partner can possible send messages ($v \not\leq : \mathbf{0}$). This ensures corollary 1 (sect.5).
SEND-ERR concerns errors while sending messages. Two cases arise: the message is not in the list of authorised sendings, and when the type is asking for a reception, or is $\mathbf{0}$.
RECV-ERR concerns reception of messages not performed.
RECV*-ERR deals with server ports in the component, when a port is not immediately created upon the reception of the request.
MUST-ERR is used when predicate $\mathrm{Must}(\mathsf{T}')$ is false (meaning: a port typed **must!** is suspended by a port typed **may?**; consequence is that the first port may not honour its contract).

4.3. *Component Honouring a Contract*

A component honouring a contract, noted $B(\mathsf{P}, \mathsf{R}, \mathsf{T}) \models \tilde{C}$, is such that the reduction process will never lead to *Error*:

$$B(\mathsf{P}, \mathsf{R}, \mathsf{T}) \models \tilde{C} \quad \text{iff} \quad \forall B', \tilde{C}' \text{ such that } (B, \tilde{C}) \rightarrow^* (B', \tilde{C}') : \quad (B', \tilde{C}') \nrightarrow \mathit{Error}$$

5. Sound Assembly of Components: Definition and Properties

So far, we defined compatibilities between a component and its interface types, and between interfaces. In this section, we investigate properties on an assembly of components,

For the sake of readability, we abbreviate:
$M_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}_k);\, T_k], M'_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}'_k);\, T'_k] \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{REMVREF-ERR}\ \frac{u\!:\!T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ T \equiv \mathbf{must?}M_\Sigma$$

$$\text{BIND-ERR}\ \frac{u\!:\!T \qquad v\!:\!S \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{bind}(u\multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg \mathit{Comp}(T,S)$$

$$\text{DEACT-ERR}\ \frac{u\!:\!T \equiv \mathit{mod}\ \mathbf{?}\ [\mathbf{*}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{deact}(u)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \begin{pmatrix} (u \multimap \bot) \\ \vee\big((u \multimap v) \wedge v \not<: \mathbf{0}\big) \end{pmatrix}$$

$$\text{SEND-ERR}\ \frac{u\!:\!T \equiv \mathit{mod}\ \rho\ [\mathbf{*}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\mathbf{!}m'} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg m'\!:\!M_\Sigma \vee \rho = \mathbf{?},\mathbf{0}$$

$$\text{RECV-ERR}\ \frac{u\!:\!T \equiv \mathit{mod}\ \mathbf{?}[\mathbf{*}]M_\Sigma \qquad \forall k, B(\mathsf{P},\mathsf{R},\mathsf{T}) \overset{u:u'\mathbf{?}m_k}{\not\longrightarrow} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}$$

$$\text{RECV*-ERR}\ \frac{u\!:\!T \equiv \mathit{mod}\ \mathbf{?*}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \overset{u:w\mathbf{?}m_k,\text{creat}(u'),\text{bind}(u'\multimap w),\text{actv}(u')}{\not\longrightarrow} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}$$

$$\text{MUST-ERR}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \rightarrow B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg\text{Must}(\mathsf{T}')$$

$$\neg m'\!:\!M_\Sigma \triangleq m' = M'(\tilde{v}') \wedge \forall k, M' \neq M_k \vee \neg\tilde{v}_k\!:\!\tilde{U}'_k$$

Table 5. *Rules for contract satisfaction (invalid transitions)*

and prove safety properties (no error occurs, and no deadlock between ports will occur), and liveness properties (all messages sent are eventually consumed).

For the sake of readability, and to abstract from component structure and its contract, we denote $u : T \in \mathsf{R}$ to indicate that, in the contracted component $B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}$: $u \in \mathsf{R}$ and $u : T \in \tilde{C}$. Also, we write $(u : T \multimap v : T')$ to state that, in $B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}$, we have $(u \multimap v) \in \mathsf{P}$ and $u\!:\!T, v\!:\!T' \in \tilde{C}$.

5.1. *Assembly of Components*

We define an assembly of components as a configuration of components with their contract, and ready to interact via a communication medium. An assembly, in its initial configuration, encompasses both client/server and peer-to-peer bindings. It has the following properties, where the last two ensure that a peer reference is private [NNS99]:

– the configuration is reference-closed: references designate ports in the configuration;
– all the ports are active on independent threads;
– peer ports in reception have a unique partner in the configuration;
– a component knows no peer references, except those of his own ports and their partners.

Definition 1 (Assembly of Components)
$\mathcal{A} = \{(B_1(P_1, R_1, T_1), \tilde{C}_1), \ldots, (B_n(P_n, R_n, T_n), \tilde{C}_n), Com\}$

$$\text{with the 4 properties: } \begin{cases} \forall i, u : & u \in R_i \Rightarrow \exists j \text{ such that } u \in P_j \\ \forall u \in \cup P_i : & T(u) = \rho^{\mathsf{a}} \\ \forall u \in \text{peer}(\cup P_i), u\mathbf{?}^{\mathsf{a}} : & \exists! v, j \text{ such that } (v \multimap u) \in P_j \\ \forall i : & u \in R_i \Rightarrow u \in P_i \ \vee \ \exists v : (v \multimap u) \in P_i \end{cases}$$

Then, a sound assembly is an assembly where each component satisfies its interface contracts, and linked ports have their interfaces mutually compatible:

Definition 2 (Sound Assembly)
$\mathcal{A}$ is sound (denoted $\models\!\!\!\!\prec \mathcal{A}$) iff
$\forall i : \big(B_i \models\!\!\!\!\prec \tilde{C}_i\big) \ \wedge \ \big((u\!:\!T_u \multimap v\!:\!T_v) \in P_i \Rightarrow (Comp(T_u, T_v) \ \wedge \ \exists!\, j, T' \text{ such that } v \mathrel{<:} T' \in P_j)\big)$

5.2. *Compatibility and Message Consumption Properties*

Compatibility relation is very important in our study. The following property ensures all references bound to port are such that their respective type are compatible.

Property 4 (Safeguarding of Compatibility)
If $\models\!\!\!\!\prec \mathcal{A}$, then $\forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C}$, we have: $\forall u, v \in \mathcal{C}, (u\!:\!T_u \multimap v\!:\!T_v) \Rightarrow Comp(T_u, T_v)$.

Proof. By structural induction, and from the definition of the compatibility relation (interfaces evolve towards compatible types). □

The next property, P_{sr}, of a sound assembly states simply that soundness is maintained throughout the evolution: a configuration of component never leads to *Error*:

Theorem 1 (Subject Reduction)
If $\mathcal{A}$ is sound, then $\mathcal{A} \models P_{\text{sr}}$, with $P_{\text{sr}} \triangleq \forall \mathcal{C} : \mathcal{A} \rightarrow^* \mathcal{C}, \mathcal{C} \not\twoheadrightarrow Error$.

Proof. By structural induction on the transition rules. The property is satisfied by observing that the only way a configuration can lead to *Error* is by violating compatibility rules. □

Corollary 1 (Message Consumption)
If $\mathcal{A}$ is sound, then $\mathcal{A} \models P_{\text{mc}}$,
with: $P_{\text{mc}} \triangleq \forall u, v, i, M : (u \multimap v) \in P_i, (\mathcal{C} \xrightarrow{u:v\mathbf{!}M} \mathcal{C}') \Rightarrow \exists \mathcal{C}'', \mathcal{C}''' : \quad \mathcal{C}' \rightarrow^* \mathcal{C}'' \xrightarrow{v:u\mathbf{?}M} \mathcal{C}'''$

Proof. This corollary is a consequence of theorem 1, the use of FIFO queues and the constraint that a port in reception is active. □

The corollary ensures that in the future, a transition will be able to consume the message. However, since the rules compete with each other, we have to assume fairness in this competition. Hence, the corollary ensures the consumption of the message in the future, but modulo fairness.

5.3. *Type of a Reference in the Configuration*

This section presents an interesting property that relates the type of a port and the type of the reference pointing at it. The following lemma gives such a relation.

Lemma 2
Given $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ *and* $B(\mathsf{P}, \mathsf{R}, \mathsf{T})$, *with* $u \notin \mathsf{R}_u$, $u{:}T_{part} \in \mathsf{R}$, *and* $Com.u = \varnothing$. *Then:*

$$(u{:}T \multimap v{:}T') \in \mathsf{P}_u \Rightarrow {T'}^{\mathcal{D}} \preceq T_{part}$$

$$(u{:}T \multimap \bot) \in \mathsf{P}_u \Rightarrow T \preceq T_{part}$$

The lemma applies only when reference u and the corresponding port are in different components ($u \notin \mathsf{R}_u$); type comparison is possible only if queues are empty, and shows that:

– if port u is bound to v: the type of reference u is a supertype of the dual of the type of v;
– if port u is unbound: the type T_{part} of reference u is a supertype of the type of u.

However, we will rather use the corollary which rises from this lemma, namely that the type T_{part} of reference u is supertype of the type of port u.

Corollary 2 (relations between the type of a reference and the type of the associated port)
$\forall u$ *such that* $Com.u = \varnothing$:

$u{:}T_{part} \in \mathsf{R} \;\wedge\; u{:}T \in \mathsf{P}_u \Rightarrow T \preceq T_{part}$;   *if* $u \in \mathsf{R}_u$ *we have* $T = T_{part}$.

Proof. Straightforward from lemma 2 and property 3.5 ($Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$). The case $u \in \mathsf{R}_u$ has two possibilities: port u has not yet been sent outside the component ($T = T_{\text{part}}$ is then obvious), or port u has been sent, and was received later on by its own component. At the time of the reception of the reference, $u : T'$, is saved with $\tilde{C} \Leftarrow u : T'$, which does not modify the type of u in $\tilde{C}$; equality $T = T_{\text{part}}$ follows. □

Proof (lemma 2, sketch).
By induction. As queues have to be empty, we consider that messages are consumed immediately. Note that the lemma is verified during the assembly, from property 3.5.

Demonstrations for rules CREAT, REMVPORT, REMVREF, DEACT and OTHER are straightforward. As for BIND, conclusion is quite obvious; concerning the bound partner: hypotheses are not changed while applying the rule. Concerning the port: the latter is sending; as we had, before the binding, $(u \multimap \bot)$, necessarily $u \in \mathsf{R}_u$ because a port which is sending cannot be sent. The lemma is then not concerned by this case.

Concerning rules SEND, RECV, RECV* and RECV-UN: we suppose a reception happens right after a sending. We have the following cases:

references $\tilde{v}_k$ *in the arguments of the message*
Reasoning is straightforward, by looking at compatibility and subtype relations: while passed through messages, references are associated to a supertype of their current type.

unknown partner at the time of the reception (rule SEND followed by RECV-UN or RECV*)
We use the dual property (Sect. 2.5). Note that, in the rule, the dual is applied only to peer references. Before applying the rules, we had (with u sending and v receiving):

$$(u:T \multimap v:T'_{\text{part}}) \qquad (v:T' \multimap \bot) \text{ with, by induction } T' \preceq T'_{\text{part}}$$

Let's write T_k, T'_{part_k} and T'_k the types resulting from the sending or receiving of message M_k. When u applies SEND, and v applies one of the RECV rules, we have:

$$(u:T_k \multimap v:T'_{\text{part}_K}) \qquad (v:T'_k \multimap u:{T'_k}^{\mathcal{D}}) \text{ with } T'_k \preceq T'_{\text{part}_k}$$

Concerning u, we use the induction hypothesis $T'_k \preceq T'_{\text{part}_k}$ and property 3.5 to prove $(T'_{\text{part}_k})^{\mathcal{D}} \preceq {T'_k}^{\mathcal{D}}$. Concerning v, $({T'_k}^{\mathcal{D}})^{\mathcal{D}} \preceq T'_{\text{part}_k}$ is straightforward (${T'_k}^{\mathcal{D}\mathcal{D}} = T'_k$).

known partner at the time of the reception (rule SEND followed by RECV)
With the same reasoning (with u sending, v receiving, w equal or different from u):

$$(u:T \multimap v:T'_{\text{part}}) \qquad (v:T' \multimap w:T'') \text{ with } {T''}^{\mathcal{D}} \preceq T'_{\text{part}}$$

When u applies SEND and v RECV, we have:

$$(u:T_k \multimap v:T'_{\text{part}_k}) \qquad (v:T'_k \multimap u:T''_k) \text{ with } {T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$$

Concerning v, ${T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$ is true. We easily deduce, for the port u: $(T'_{\text{part}_k})^{\mathcal{D}} \preceq T''_k$ from $({T''_k}^{\mathcal{D}})^{\mathcal{D}} = T''_k$ and property 3.5. □

5.4. *External Deadlock Freeness*

External deadlock represents the situation where a set of ports are inter-blocked because of a dependency cycle. The simplest form of external deadlock is written:

$$(u \multimap u') \wedge (v \multimap v') \wedge (u\mathbf{!}^{\mathsf{s}} \rightarrowtail v\mathbf{?}^{\mathsf{a}}) \wedge (v'\mathbf{!}^{\mathsf{s}} \rightarrowtail u'\mathbf{?}^{\mathsf{a}})$$

u sending is blocked by v which is waiting for v' to send which, in turn, is blocked by u', which is waiting for u to send.

The general case can be formalised, on the principle of *Wait-For-Graph*, as the existence of cycles in a dependency graph. We introduce a new dependency relation between the ports, named external dependency, denoted $\cdot \dashrightarrow \cdot$. This dependency relates the communications between distant ports. For example, $u\mathbf{?}^{\mathsf{a}} \dashrightarrow v\mathbf{!}^{\mathsf{s}}$ means u depends on v, more precisely u waits for v to send.

Definition 3 (External dependency $\dashrightarrow$)
$u \dashrightarrow v$ *iff* $u\mathbf{?}^{\mathsf{a}} \wedge \big((v\mathbf{!}^{\sigma} \wedge (v \multimap u)) \vee (v\mathbf{?}^{\mathsf{a}} \wedge (u \multimap v) \wedge u : M(\ldots) \in Com.v)\big)$

Remarks: if u and v are both typed **may?**, all queues are empty and there is no dependency. In the other cases, the fact that v can be in reception ($u\mathbf{?}^{\mathsf{a}} \dashrightarrow v\mathbf{?}^{\mathsf{a}}$) is a particular case which is due to asynchronous messages: this case shows that u has just sent a message M and waits a response from v, whereas v has not yet consumed M.

A deadlock is then formalised as follow:

Definition 4 (Ext_deadlock)
$\text{Ext_deadlock}(\mathcal{C}) \triangleq \exists (u_i)_{1..n} \in \mathcal{C}$ *such that* $\forall k < n : u_i \mathcal{S} u_{i+1} \wedge u_n \mathcal{S} u_1$
with $u\mathcal{S}v = true$ *when* u *is suspended to* v: $\quad u\mathcal{S}v \triangleq u \rightarrowtail v \;\vee\; u \dashrightarrow v$

Note that idle or active ports in a sending state are not concerned by a deadlock.

Theorem 2 (External deadlock freeness)
If $\mathcal{A}$ is sound, then $\mathcal{A} \vDash P_{\mathrm{edf}}$ with $P_{\mathrm{edf}} \triangleq \forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C} \Rightarrow \neg \mathrm{Ext_deadlock}(\mathcal{C})$

The deadlock freeness problem has received attention recently. A work on this issue which is very close to ours is the one by Naoki Kobayashi [Kob02], where the author does have **may** and **must** actions (in terms of capabilities and obligations), but communications are synchronous, and constraints between compatible types are too restrictive for our study.

Proof of theorem 2 is tedious. Even if interfaces are mutually compatible, it is not straightforward that a deadlock will not arise between components (ports in a component may be suspended by other ports, which leads to potential dependencies between threads added to dependencies between ports). The proof needs two lemma, presented hereafter. The first one indicates where a peer reference is.

Lemma 3
Given a peer reference v, one and only one of the following alternatives hold:

1) if $\exists B(\mathsf{P},\mathsf{R},\mathsf{T})$ such that $v \in \mathsf{R}$, then B is unique;

2) $\exists!(u, M)$ such that $M(\ldots, v, \ldots) \in Com.u$; moreover $\nexists B(\mathsf{P},\mathsf{R},\mathsf{T})$ such that $v \in \mathsf{R}$;

3) $\exists M(\ldots)$ such that $v : M(\ldots) \in Com.u \wedge (u \not\multimap v)$; moreover $\nexists B(\mathsf{P},\mathsf{R},\mathsf{T})$: $v \in \mathsf{R}$;

Proof. By structural induction. We present here the non obvious cases.

• *Rule SEND*: u sends message $M_k(\tilde{v})$ towards u'. We will use the notation $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ for a component for each $u \in \mathsf{P}_u$. We consider three kinds of references:

a peer reference $v \in \tilde{v}$ is in an argument of a message
By induction, B_u is the unique component knowing v ($v \in \mathsf{R}_u$); when v is sent, it is removed from R_u. Hence, after applying SEND, v is known of no component (point 1. of the lemma is false), and there is a unique message in Com such that v is in the arguments of the message (point 2. of the lemma is true). Point 3. of the lemma is also false.

reference u' which receives the message
Lemma remains true for u': only the first point of the lemma is true ($u' \in \mathsf{R}_u$).

reference u that sent the message
Considering the case where u' is bound to u, we have $u \in \mathsf{R}_{u'}$, and points 2. and 3. of the lemma are false. Considering the case where $(u' \not\multimap u)$, two possibilities arise:

u is sending for the first time towards u'. First, let's look at the binding of u' to port u. It may be due either to a reception of a message or to the BIND rule. The first case is impossible because $(u' \not\multimap u)$, so u' could not have send the message. Hence, $(u \multimap u')$ is due to the BIND rule, and no reception has occurred since. As u is sending for the first time, u has made no action between BIND and SEND. Thus, we know that since u has been created, it was always in a sending state. As this is its first sending, we have, before applying SEND: $u \in \mathsf{R}_u$, thus only point 1. of the lemma is true. After the sending, only point 3. of the lemma is true.

u has already sent a message towards u'. Point 3. of the lemma was true, which remains true after the sending (note that the first message that u sent is not yet consumed by u', because $(u' \not\multimap u)$).

• *Rules RECV, RECV*, RECV-UN* (sketch): reasoning is the same as for rule SEND: we give here only a sketch. Considering message $u' : M(\tilde{v})$ received by u, we have three cases:

references as argument of the received message
: Concerning those references, point 3. of the lemma is true before the reception; only point 1. is true after the reception.

the reference u' that sent the message
: In the case where $(u \multimap u')$ before the reception, then there is no modification (only point 1. of the lemma is true). In the case where u' is a new partner for u, then point 3. of the lemma was true before the reception, and only point 1. is true after.

reference u that receives the message
: There are no changes during the application of the rule. □

The second lemma concerns external dependency relations between peer ports. It shows that this relation is unique. It is a consequence of lemma 3.

Lemma 4
If u and v are peer ports such that $u \dashrightarrow v$, then u and v are unique.

Proof. Given u and v such that $u \dashrightarrow v$. By definition:

$$u\mathbf{?}^{\mathsf{a}} \wedge \big((v\mathbf{!}^{\sigma} \wedge (v \multimap u)) \vee (v\mathbf{?}^{\mathsf{a}} \wedge (u \multimap v) \wedge u : M(\ldots) \in Com.v)\big)$$

We have two cases (we denote $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ the component B_u such that $u \in \mathsf{P}_u$):

v is sending: $v\mathbf{!}^{\sigma} \wedge (v \multimap u)$.
: As v can have only one partner, there is only one u such that $u \dashrightarrow v$. Concerning the unicity of v: as we have $(v \multimap u)$, necessarily $u \in \mathsf{R}_v$ (by looking at how a binding is made – through BIND and RECV rules – and also because a bounded reference cannot be sent). Suppose there exists $B_w(\mathsf{P}_w, \mathsf{R}_w, \mathsf{T}_w)$ whereby $w \neq v$ and $(w \multimap u)$; then we have $u \in \mathsf{R}_w$. Lemma 3 ensures $B_w = B_v$. As it is not possible to have, in the same component, a reference bound to two different ports, we have $w = v$ and v is unique.

v is receiving: $v\mathbf{?}^{\mathsf{a}} \wedge (u \multimap v) \wedge u : M(\ldots) \in Com.v$
: The same reasoning applies (suppose there exists $w \neq u$ having the same properties than u, then $w = u$ necessarily). □

The latter lemma is used in the demonstration of theorem 2:

Proof (External Deadlock Freeness).
By structural induction on the contract rules. CREAT, REMVPORT, REMVREF and DEACT do not create any dependency relation, thus are not concerned. As for BIND, a dependency relation is created, but as only active or idle ports in a sending state can apply this rule, and as those ports are not concerned by predicate $\mathrm{Ext_deadlock}(\mathcal{C})$, no deadlock is created.

• *Rule SEND*: We distinguish between the next action ρ of port u:

$\rho = \mathbf{!}$ dependency relations remain unchanged;

$\rho = \mathbf{0}$ dependency between u and its partner disappears;

$\rho = \mathbf{?}$ this is the particular case $u\mathbf{?}^{\mathsf{a}} \dashrightarrow v\mathbf{?}^{\mathsf{a}}$, with v the partner of u: $(u \multimap v)$.
: *Reductio ad absurdum*: suppose this dependency created a deadlock:

$$\exists (u_i)_{1..n} \in \mathcal{C} \text{ such that } \forall k : u_i \mathcal{S} u_{i+1} \text{ with } u, v \in (u_i)$$

Let's chose (u_i) such that $u_1 = u$ and $u_2 = v$ (as we have $u\mathcal{S}v$) and let's show that if $v\mathcal{S}w$ then $w = u$. As v is active, the only dependency between u and v is an external one, and by lemma 4 we have $w = u$. Hence we have, simultaneously:

$$u \dashrightarrow v \text{ that is to say } v\mathbf{?}^{\mathsf{a}} \;\wedge\; (u \multimap v) \;\wedge\; u : M(\ldots) \in Com.v$$
$$v \dashrightarrow u \text{ that is to say } u\mathbf{?}^{\mathsf{a}} \;\wedge\; (v \multimap u) \;\wedge\; v : M'(\ldots) \in Com.u$$

In particular, we have $v : M'(\ldots)$ in $Com.u$, which is contradictory since u has just sent a message, and types of u and v are compatible (so u would have made more receptions that the number of sendings from v).

• *Rules RECV, RECV-UN* (sketch): as for SEND, we have three cases, depending on ρ:

$\rho = \mathbf{?}$ if an external dependency arises, it is identical to the one before the reception.

$\rho = \mathbf{0}$ dependency between u and its partner disappears;

$\rho = \mathbf{!}$ then we have $\mathsf{T}(u) = \mathbf{!}^{\mathsf{a}}$. u is not concerned by $\text{Ext_deadlock}(\mathcal{C}')$, hence the predicate is false (by induction).

• *Rule RECV**: Upon the reception of the message, a new port u' is created, with its own thread of execution. Hence, there is a new external dependency, but no deadlock arises.

• *Rule OTHER*: This rule concerns other transitions of the component, namely C-ACTV and C-ACTV2. The rule C-ACTV adds dependency $u\mathbf{!}^{\mathsf{s}} \rightarrowtail v\mathbf{!}^{\mathsf{a}}$; the port v is not concerned by predicate $\text{Ext_deadlock}(\mathcal{C}')$. Conclusion for C-ACTV2 is straightforward, as a port is activated on its own thread of execution. □

5.5. *Liveness Properties under Assumptions*

The assembly of components may have still a livelock problem: a port can be forever suspended because of a divergence of some internal computation or an endless dialogue between two ports. Thus it is not possible to prove a liveness property that states "each port reaching a **must?** (or **must!**) state will eventually receive (or send) a message":

$$P_{\text{must}} \triangleq \forall\, \mathcal{C}, u, i : \;\; \mathcal{A} \rightarrow^* \mathcal{C}, (u\!:\!\textbf{must}\rho M_\Sigma) \in \tilde{C}_i \text{ with } \rho \in \{\mathbf{?}, \mathbf{!}\} \Rightarrow$$
$$\exists \mathcal{C}', \mathcal{C}'', v \text{ such that } \mathcal{C} \rightarrow^* \mathcal{C}' \xrightarrow{u:v\rho M_k} \mathcal{C}''$$

However, we believe this liveness property is verified with the assumptions:

– a computation in a component always ends;

– a suspended port which becomes active must send its message before suspending again;

– a port which has a loop behavior will become idle in the future.

Anyhow, these properties can only be verified at a lower level of abstraction, that is, only when the concrete behaviour of the components (e.g. its source code) is known.

6. Bank Account Example

To illustrate our framework, we use yet another bank account example: a client has to authenticate himself to the bank, so he can perform deposit and withdrawal operations on his bank account. We propose an implementation with three components: the `Client`, the `Bank` and the `Account`.

Several possible implementations exist; for example the `Client` receives a reference to his `Account` so he can interact with it. We propose a more elegant solution, described by figures 2 and 3: the `Client` will use the same port for the authentication and the operations on his account. The `Bank` identifies the client (figure 2: the client was correctly authenticated), and sends to the `Account` the reference of the port of the `Client`. `Account` will use this reference to inform the `Client` that on the one hand access is granted, and on the other hand the dialogue continues with `Account`. Figure 3 shows that if the user is not correctly identified, the `Bank` will return the `refused` message. The salient issue of this protocol is that the couple (`Bank`, `Account`) can be replaced by only one component, *without any change of* `Client`.

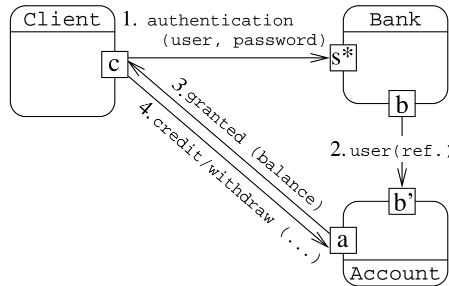


Figure 2. *Example: access to bank account granted*

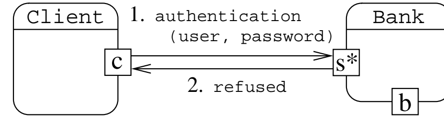


Figure 3. *Example: access to bank account refused (component `Account` omitted)*

6.1. *Interface Types*

The `Bank` component has two ports: s^* is a server port for authentications, and b is used to delegate operations to the `Account` component. The type of s^* is described hereafter:

bank_access = **may ? *[** *authentication* (string, string)**; must ! [** *granted* (real)**;** operations
+ *refused***; 0]]**

This server type can receive the *authentication* message (with user and password); after receiving this message, the port must either send *granted* with the balance and become *operations*, or send *refused* and stop. Type *operations* is described further.

In the case where the `Client` is correctly authenticated, the `Bank` delegates the sending of the *granted* message to the `Account`. This is done via port b, which is specified by the type *send_ref*. This type must send a reference which is typed *granted_user*: this latter type must receive only the *granted* message, and then become *client_operations*, the dual of *operations*.

send_ref = **must ! [** *user* (granted_user)**;** send_ref **]**
granted_user = **must ? [** *granted* (real) **;** client_operations **]**
client_operations = operations$^{\mathcal{D}}$

The `Account` component has also two ports: b' and a. The former is used to receive the reference of the client (type *received_ref*). The latter port, a, will interact with the `Client`. His type *access_granted* is written hereafter.

receive_ref = **may ? [** *user* (granted_user)**;** receive_ref **]**
access_granted = **must ! [** *granted* (real) **;** operations **]**

The type *operations*, which specifies deposits and withdrawals, is described in two times:

operations = **may ? [** *deposit* (real)**;** **must ! [** *balance* (real)**;** operations **]**
+ *withdraw* (real)**; must ! [** *balance* (real)**;** operations
+ *neg_balance* (real)**;** negbal_operations **]]**
negbal_operations =
must ? [*deposit* (real)**;** **must ! [** *balance* (real)**;** operations
+ *neg_balance* (real)**;** negbal_operations **]**
+ *withdraw* (real)**; must ! [** *neg_balance* (real)**;** negbal_operations **]]**

A port typed *operations* can receive two messages: *deposit* and *withdraw*. After receiving one of the two messages, the port must send back the balance of the bank account: message *balance* is sent when balance is positive, and the type becomes *operations* again. Message *neg_balance* is sent when the user is debtor, and the type becomes *negbal_operations*.

The difference between the two types *operations* and *negbal_operations* is the modality: *operations* **may** receive messages, whereas *negbal_operations* **must** receive messages. Hence, as long as the client is debtor, he must perform some operation on his bank account.

Finally, the `Client` has one port, *c*, whose type can be written as follows. With this type, the `Client` will try to perform one operation. However, if this operation results in a negative balance, then other operations have to be made.

client = **must ! [** *authentication* (string, string)**;**
must ? [*granted* (real)**;** client_simple_operation
+ *refused***; 0]]**
client_simple_operation =
must ! [*deposit* (real)**;** **must ? [** *balance* (real)**; 0**
+ *neg_balance* (real)**;** client_simple_operation **]**
+ *withdraw* (real)**; must ? [** *balance* (real)**; 0**
+ *neg_balance* (real)**;** client_simple_operation **]]**

We have the following relations between the types:

Compatibilities to be checked during assembly	$Comp$(client, bank_access) $Comp$(send_ref, receive_ref)
Other compatibilities	$Comp$(granted_user, access_granted) $Comp$(client_waitauth, access_granted) $Comp$(client_simple_operation, operations) $Comp$(client_simple_operation, negbal_operations)
Other relations	client_waitauth $\preceq$ granted_user client_simple_operation $\preceq$ client_operations granted_user = access_granted$^{\mathcal{D}}$

with: client_waitauth = **must ? [** *granted* (real)**;** client_simple_operation **+** *refused***; 0]**

6.2. *Component Specification and Contract Verification*

For lack of space, we present only the most interesting component, that is `Bank`. Figure 4 shows in one shot the specification and the contract verification of the `Bank` component.

The specification is shown as simple state-charts (each bold rectangle representing a state $B(\mathsf{P}, \mathsf{R}, \mathsf{T})$. The greyed parts represent the changes compared to the previous state); transitions are labelled with actions as named in the rules. Reception of message *authenticate(...)*, in the upper part of the diagram, makes the component create a new port, s', to answer the client, which corresponds to the three states *Create Child*. This sequence terminates with "$\mathrm{actv}(s')$", which has two effects:

– go back to the state *Access* to answer new requests;

– create a new thread of execution. This thread will identify the user (state *Check Access*). If the user is not correctly identified (left branch), message *refused* is sent, and ports removed. If the user is correctly identified (right branch), port b is activated and sends message *user(c)* towards port b' of the `Account` component; note that c cannot be sent if bound to s', so we first have to deactivate and remove port s'. Also, as c is a peer reference, it has to be removed from R when sent to b'.

Contracts associated to each state are shown with parallelograms. Greyed parts show the verifications that are made. Once message *authenticate(...)* is received, the component performs the correct sequence of actions. The contract associated with the *Check Access* state contains both the type of the server port s^* and the type of the created port s'. Note also an application of corollary 2: the type of reference c (*client_operations*) is the dual of s', and is a super-type of the type of port c (*client_simple_operation*).

7. Conclusion & Future Work

We have presented a concept of behavioural contracts that we applied on a component model featuring multiple threads, reference passing, peer-to-peer and client/server communication patterns. Our contracts serve for the early verification of compatibility between components, in order to guarantee safety and liveness properties. Compatibility is formally described in this framework, as a composition of internal compliance of components to their interfaces, and conformance between interfaces.

In the context of component based design, the verification that a component is honouring a contract given by its interfaces is in charge of the component producer, which performs it once for all. A certification of this fact may be produced by some certification authority, in order for example to guarantee any recipient of a publicly available or migrating component that the component does not do anything different but what is described by its interfaces.

The verification of interface compatibility should instead be performed when the component is bound to another (e.g. at run-time when dealing with migrating code, that is when a migrating component reaches its final destination). We have shown that this check can be performed very efficiently by means of standard finite state space verification techniques. The higher complexity of checking conformance of components to their declared interface is left to an off-line verification activity, which may even need the use of infinite-state space verification techniques.

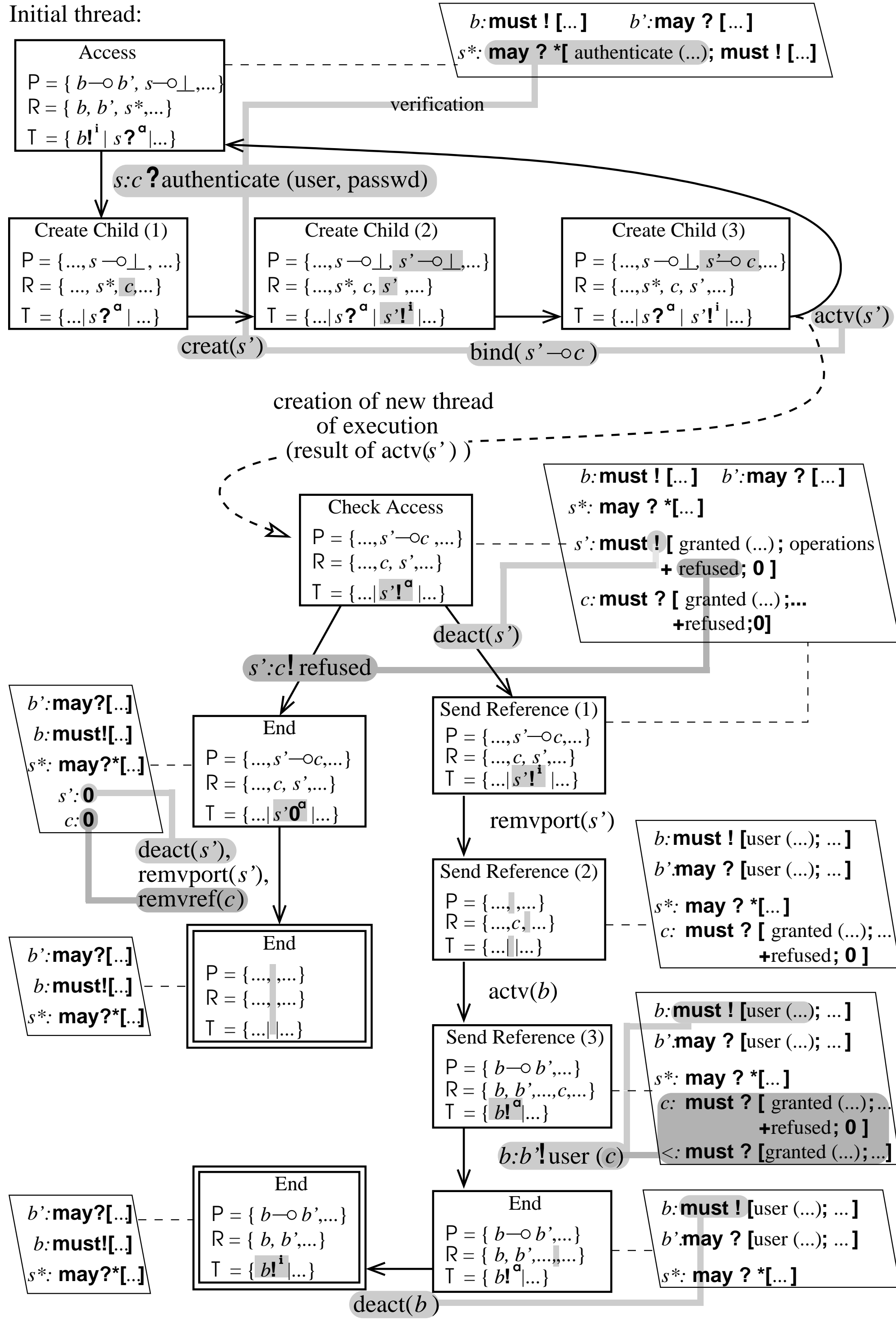


Figure 4. *Specification and verification of the* `Bank` *component*

We have applied our approach to a toy example; we need to verify the usability of the approach in practice, especially with respect to the expressiveness of the interface language we have proposed. Mixing modalities and actions (send/receive) is under investigation at NTNU, and may lead to some constraints like the ones proposed in [Flo03]. The conformance of the component model we have assumed with concrete notations (e.g. Java) should also be studied: varying the component model to suite a concrete notation may actually affect the classes of properties that can be guaranteed. We can also observe that the compatibility rules can be expressed in terms of temporal logic formulae: this would make it possible to prove in a logical framework a richer set of properties.

Acknowledgements C. Carrez and E. Najm have been partially supported by the RTNL ACCORD project and by the IST MIKADO project[6]. A. Fantechi has been partially funded by the 5% SP4 project of the Italian Ministry of University and Research. E. Najm has also been partially supported by a grant from ISTI of the Italian National Research Council.

Special thanks to Arnaud Bailly for his helpful advices.

8. References

[Car03] C. Carrez. *Contrats Comportementaux pour Composants*. PhD thesis, ENST, Paris, France, December 2003.

[CFN03a] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound composition of components. In *Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6/WG 6.1)*, volume 2767 of *LNCS*. Springer-Verlag, Berlin, Germany, September 2003.

[CFN03b] C. Carrez, A. Fantechi, and E. Najm. Contrats comportementaux pour un assemblage sain de composants. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2003)*, Paris, France, October 2003. French version of [CFN03a].

[dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-01*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*. ACM Press, 2001.

[Flo03] J. Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, NTNU, Trondheim, Norway, February 2003.

[HR02] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *INFCTRL: Information and Computation (formerly Information and Control)*, 173, 2002.

[Kob02] N. Kobayashi. A type system for lock-free processes. *INFCTRL: Information and Computation (formerly Information and Control)*, 177, 2002.

[KPT99] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

[LSW95] K.G. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology based on modal transition sytems. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS'95*, volume 1019 of *LNCS*, 1995.

[Nie95] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[NNS99] E. Najm, A. Nimour, and J.-B. Stefani. Infinite types for distributed objects interfaces. In *Proc. of Formal Methods for Open Object-based Distributed Systems - FMOODS'99*, Firenze, Italy, February 1999. Kluwer.

---

6. Resp. `http://www.infres.enst.fr/projets/accord` and `http://mikado.di.fc.ul.pt/`