Behavioural Contracts for a Sound Assembly of Components

Cyril Carrez[1], Alessandro Fantechi[2,3], and Elie Najm[1]

[1] Ecole Nationale Supérieure des Télécommunications, Département INFRES,
46 rue Barrault, F-75013 Paris, France,
{cyril.carrez, elie.najm}@enst.fr
[2] Universitá di Firenze, Dipartimento di Sistemi e Informatica,
Via S. Marta 3, I-50139 Firenze, Italy,
fantechi@dsi.unifi.it
[3] ISTI – CNR,
Via G. Moruzzi 1, I-56124 Pisa, Italy

Abstract. Component based design is a new methodology for the construction of distributed systems and applications. In this new setting, a system is built by the assembly of (pre)-existing components. Remains the problem of the compositional verification of such systems. We investigate methods and concepts for the provision of "sound" assemblies. We define an abstract, dynamic, multi-threaded, component model, encompassing both client/server and peer to peer communication patterns. We define a behavioural interface type language endowed with a (decidable) set of interface compatibilty rules. Based on the notion of compliance of components to their interfaces, we define the concepts of "contract" and "contract satisfaction". This leads to the notion of sound assemblies of components, i.e., assemblies made of contracted components interacting through compatible interfaces. Sound assemblies possess interesting properties like "external deadlock freeness" and "message consumption".

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

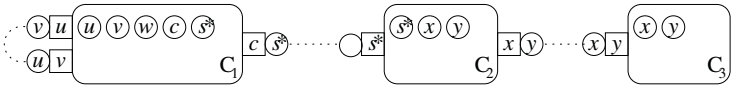
H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 111–126, 2003.
© IFIP International Federation for Information Processing 2003

formal description of the software artifact he is building, but gets a guarantee about the absence of mismatches between components from the underlying formally verified components and from 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.

Our work is in part inspired by the work by 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 limiting 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 Sect. 2 we show the reference component model on which we base the definition (Sect. 3) of our interface language and the related compatibility rules. In Sect. 4 we give the concept of *component honouring a contract*. Sect. 5 describes the properties that can be guaranteed by sound assemblies of components.



$\mathsf{R}_1 = \{u, v, w, c, s^*\}$ $\mathsf{P}_1 = \{(u \multimap v), (v \multimap u), (c \multimap s^*)\}$
$\mathsf{R}_2 = \{s^*, x, y\}$ $\mathsf{P}_2 = \{(s^* \multimap \bot), (x \multimap y)\}$ $\mathsf{R}_3 = \{x, y\}$ $\mathsf{P}_3 = \{(y \multimap x)\}$

Fig. 1. An example of a configuration

2 Component Model

2.1 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 configuations: a component may create new ports and may also dynamically bind a partner reference to any of his 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. 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 ***** is used to indicate 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).

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 respectively when the head port is suspended and the activity is passed 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 behavioral 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 behavioral notation for components.

2.2 Notations for Components

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

The set of references is noted R, and is ranged over by u, v, w, c, s. Classical set notation is used for the operations on R; however, we use the unorthodox $R \cup u$ notation for the insertion of an element.

The set of ports is noted P. We will, in fact, consider P as a set of mappings from port references to partner references. We note $(u \multimap v)$ the mapping of port u to partner v – a port which has no partner is written $(u \multimap \bot)$. The following notations will be useful for the manipulation of port mappings:

- $\mathsf{P}[u \multimap \bot]$ port u is added to P.
- $\mathsf{P}[u \multimap v]$ attach the partner v to port u. Overrides the previous partner.
- $\mathsf{P}\backslash u$ remove the port u from P.
- $(u \multimap v) \in \mathsf{P}$ port u is in P, and is attached to v. We'll write also $u \in \mathsf{P}$ to check only the membership of u to P.

The set of threads, T, reflects the state of the ports of the component and the dependencies between them. The state of a port present in P is abstracted to one of : activated or suspended or idle, and one of : sending or receiving or no action. We formally denote the activity state, $u\rho^\sigma$, of a port u, as follows:

$$\rho = \begin{cases} \mathbf{!} & u \text{ is in a sending state} \\ \mathbf{?} & u \text{ is in a receiving state} \\ \mathbf{0} & u \text{ has no action} \end{cases} \quad \text{and } \sigma = \begin{cases} \mathsf{a} & u \text{ is active} \\ \mathsf{s} & u \text{ is suspended (by a port)} \\ \mathsf{i} & u \text{ is idle} \end{cases}$$

We do not allow the combination $u\mathbf{?}^{\mathsf{s}}$, reflecting that a port waiting for an input is always active. $u\mathbf{?}^{\mathsf{a}}$ is an active port waiting for a message. The behavior of $u\mathbf{!}^{\mathsf{a}}$ is that it can either send a message or become suspended by another port. The only allowed behavior for $u\mathbf{0}^{\mathsf{a}}$ is to give back the thread of control, become $u\mathbf{0}^{\mathsf{i}}$ and vanish. 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 $x_1 \rightarrowtail x_2 \rightarrowtail \cdots \rightarrowtail x_n$. This sequence has the following constraints:

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

The following operations on T are defined (with $x = u\rho^\sigma$ occuring only once in T):

- $\mathsf{T} \mid x$ add a port with its own thread of execution (i.e. no dependency).
- $\mathsf{T}\backslash u$ this operation is defined only if u is the head of some thread $t \in \mathsf{T}$. Remove u from t and puts the port next the head in active state.
- $\mathsf{T}[u \rightarrowtail v]$ 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'.
- $\mathsf{T}[u\rho^{\sigma \to \sigma'}]$ changes the activity of a port.
- $\mathsf{T}(u)$ returns ρ^σ if $u\rho^\sigma \in \mathsf{T}$.

2.3 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 reference 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 reference u.
$Com.u$	the queue for reference u. It is an ordered set of messages of the form $v : M(\tilde{w})$ where v is the reference of the sending port, M is the name of the message, and $\tilde{w}$ its arguments.
$Com \backslash u$	the u queue 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.

2.4 Component Semantics

A component is defined by: $C = B(\mathsf{P}, \mathsf{R}, \mathsf{T})$, where:

B is the state of the component.

$\mathsf{P}, \mathsf{R}, \mathsf{T}$ are the ports, references and threads as defined previously.

The rules in Tab. 1 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 first two rules describe the relation between the component and Com, for what concerns sending and receiving of messages: the message is put in, or removed from, the proper queue. CCREAT and CREMV rules describe the creation and deletion of a port (which imply the creation/removal of the corresponding queue in Com). CBIND and CUNBIND are used to respectively attach and detach a partner reference to a port, thus linking a partner port to a local one. Finally, CACTV and CACTV2 describe how a port v is activated, respectively when u is suspended by v, or v has its own thread of execution. DEACTV deactivates a port (i.e. makes it become idle).

2.5 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.

We give in Tab. 2 the communication rule for a configuration with two components, in which it is evident that the communication is not synchronous, but through the Com medium abstraction. Extension to configurations with more components is straightforward.

Table 1. Rules for component semantics

$$\text{CSEND}\ \frac{\mathsf{R}' \subseteq \mathsf{R} \qquad \mathsf{T}' = \mathsf{T}[u!/u\rho] \qquad Com' = Com[u' \triangleleft u : M(\tilde{v})]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u'!M(\tilde{v})} B'(\mathsf{P},\mathsf{R}',\mathsf{T}'), Com'}\triangle$$

$$\text{CRECV}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap u'] \qquad \mathsf{R}' \subseteq \mathsf{R} \cup \{\tilde{v}, u'\} \qquad \mathsf{T}' = \mathsf{T}[u?/u\rho] \qquad Com' = Com[u\triangleright]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u'?M(\tilde{v})} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\triangledown$$

$$\text{CCREAT}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap \bot] \qquad \mathsf{R}' = \mathsf{R} \cup u \qquad \mathsf{T}' = \mathsf{T} \mid u\rho^{\mathsf{i}} \qquad Com' = Com[\triangleleft u]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\, u \notin \mathsf{P} \wedge Com.u = \bot$$

$$\text{CREMV}\ \frac{\mathsf{P}' = \mathsf{P}\backslash u \qquad \mathsf{R}' = \mathsf{R}\backslash u \qquad \mathsf{T}' = \mathsf{T}\backslash u \qquad Com' = Com\backslash u}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\square$$

$$\text{CBIND}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P}',\mathsf{R},\mathsf{T}), Com}\, v \in \mathsf{R} \wedge (u \multimap \bot)$$

$$\text{CUNBIND}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap \bot]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P}',\mathsf{R},\mathsf{T}), Com}\, (u \multimap v) \wedge T(u) = \rho^{\mathsf{i}}$$

$$\text{CACTV}\ \frac{\mathsf{T}' = \mathsf{T}[u \rightarrowtail v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\diamondsuit$$

$$\text{CACTV2}\ \frac{\mathsf{T}' = \mathsf{T}[u\rho^{\mathsf{i}\to\mathsf{a}}]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\, \mathsf{T}(u) = \rho^{\mathsf{i}}$$

$$\text{CDEACT}\ \frac{\mathsf{T}' = (\mathsf{T}\backslash u) \mid u\rho^{\mathsf{i}}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \to B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\, \mathsf{T}(u) = \rho^{\mathsf{a}} \wedge \rho \neq ?$$

$\triangle \triangleq (u \multimap u') \in \mathsf{P} \ \wedge\ \mathsf{T}(u) = !^{\mathsf{a}} \ \wedge\ \tilde{v} \subseteq \mathsf{R} \ \wedge\ (\forall v \in \tilde{v} \cap \mathsf{P} : \mathsf{T}(v) = ?^{\mathsf{a}}) \ \wedge\ u' \in Com$

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

$\square \triangleq Com.u = \varnothing \wedge \mathsf{T}(u) = \rho^{\mathsf{i}}$ for some ρ

$\diamondsuit \triangleq \mathsf{T}(u) = !^{\mathsf{a}} \ \wedge\ \mathsf{T}(v) = \rho^{\mathsf{i}} \ \wedge\ (v \not\multimap \bot)$

Table 2. Rules for Configurations of Components

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

3 Interface Types

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

3.1 Syntax of the Interface Language

The interface language has the following syntax:

$$
\begin{array}{rcl}
\mathit{type} & ::= & \mathit{server\_name} = \mathit{mod}\ \mathit{receive}^* \\
& | & \mathit{peer\_name} = (\mathit{mod}\ \mathit{send} \mid \mathit{mod}\ \mathit{receive}) \\
\mathit{send} & ::= & \mathbf{!}\ [\ \sum_i M_i; I_i\ ] \\
\mathit{receive} & ::= & \mathbf{?}\ [\ \sum_i M_i; I_i\ ] \\
I & ::= & \mathbf{0} \mid \mathit{peer\_name} \mid \mathit{mod}\ \mathit{send} \mid \mathit{mod}\ \mathit{receive} \\
\mathit{mod} & ::= & \mathbf{may} \mid \mathbf{must} \\
M & ::= & \mathit{name}\ (\ \widetilde{\mathit{args}}\ ) \\
\mathit{args} & ::= & \mathit{peer\_name} \mid \mathit{server\_name}^*
\end{array}
$$

The **!** and **?** keywords are the usual sending and receiving actions. The modalities **may** and **must** distinguish between permissions and obligations for the performance of the actions. The choice operator **+** allows to choose one message among the list, and the **;** is used to sequence behaviors. The meaning of modalities is:

- **may ?** ΣM_i means "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".
- **must ?** ΣM_i means "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".
- **may !** ΣM_i means "the port may send to the partner any of the messages M_i, and the partner must be ready to receive it"
- **must !** ΣM_i means "the port guarantees to send one of the M_i messages to its partner, and the partner must be ready to receive it".

Messages contain arguments. Thus, references to ports, be it *peer_name* or *server_name*, can be passed in messages. Our type language does not cater for *basic_type* values (as integers, floats, ...), but their addition is straightforward.

Sending or receiving references implies some restrictions that are enforced on the behavior of the involved components:

! m(I) means "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 **?**."[1]

? m(I) means "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 **?**."

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

$I = mod$ **? [** $m()$**;** I' **]*** after the reception of m, a port whose behavior is I' is created while the server is regenerated as I. The new port will interact with the sender of the request.

For example the interface definition: ex **= must ! [** $m_1()$**;** I_1 **+** $m_2(I_2)$**;** ex **]** means that the interface *will* send either a message m_1 or a message m_2. In the first case the interface becomes another interface (type) I_1, while in the other a reference of type I_2 is sent, and the control goes back to the interface itself.

An HTTP server can be written using the spawning syntax:

http_serv **= may ? [** BasicRequest (string)**; must ! [** Response (string)**; 0]**
+ CGIRequest (string)**;** HandleCGI **]***

Upon reception of a simple request, the server creates a port which will send the response back to the client; while upon reception of a CGI request the server will create a port whose behavior is described by *HandleCGI*. In both cases, the server will become *http_serv* after receiving the requests.

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 Sect. 3.2

The interface language we defined above has several limitations. First of all, it is not possible to send and receive messages on a port at the same time. However, a work-around we can propose is to instantiate two ports: one which will deal with receptions, and one for the sendings.

A second limitation is the fact that we cannot mix **may** and **must** modalities, for example $I = ($**must ?** $M)+($**may !** $N)$. Mixing modalities rises a problem of fairness: in this example, the associated component may never consume M, just because it is still busy with sending N. To avoid this, we should insert some QoS constraints, stating for example that "**must ?** M" has to be honoued in a 5-time delay. The time constraint can be either related to a time domain (as in Arnaud Bailly's timed constraint π-calculus [Bai02]), or based on the number of reductions as in [Kob02]. Future work on this topic should take into account the need to maintain as low as possible the complexity of interface compatibility verification: this is the principle that has suggested the limitations themselves.

[1] This constraint is inevitable: if the first action of I is **!**, then a message may be sent to a third port, and will lead to incompatible behaviours between components.

3.2 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 (supposing that if one is sending, the other is receiving):

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

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 $Comp_{\text{mod}}(mod_I\ [\mathbf{!}|\mathbf{?}], mod_J\ [\mathbf{!}|\mathbf{?}])$. Its truth table is reproduced hereafter:

	I				
J	**must ?**	**may ?**	**must !**	**may !**	**0**
must ?			√		
may ?			√	√	√
must !	√	√			
may !		√			
0		√			√

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 syntactically equal with each other[2]. This is formally defined:

$$\begin{aligned} Comp_{\text{msg}}(M, M') &\triangleq Comp_{\text{msg}}(M(I_1, \ldots, I_n), M'(J_1, \ldots, J_m)) \\ &\triangleq M = M' \wedge n = m \wedge \forall i, I_i = J_i \end{aligned}$$

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):

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

For example, an HTTP client which is compatible with *http_serv* :

[2] We could use a subtype relation, which for the lack of space we do not include here.

client = **must ! [** BasicRequest (string)**; must ? [** Response (string)**; 0]]**

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.

4 Contract Satisfaction

The interface language presented in the previous section 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 contract described by this language. The definitions of Sect. 2 are extended with the notion of contract. A component has a set of contracts, one for each port. We use the notation:

- $u : T$ reference u has the contract behaviour T, which is a *type* (Sect. 3).
- $(B, \tilde{U})$ B has the contracts $\tilde{U}$, a set of $(u : T)$, such that each reference (ports and partners) has a contract associated. Addition or update of a reference is denoted $\tilde{U} \Leftarrow (u' : T')$, and removal $\tilde{U} \backslash u$.

In the following, for the sake of readability, we abbreviate: $M_\Sigma = \mathbf{[}\Sigma_k M_k(\tilde{T}'_k)\mathbf{;}\ T_k\mathbf{]}$, $M_\Sigma\mathbf{*} = \mathbf{[}\Sigma_k M_k(\tilde{T}'_k)\mathbf{;}\ T_k\mathbf{]*}$, and $m_k = M_k(\tilde{v}_k)$. We also write Must(T) a predicate stating that any reference u of the thread T which is typed **must !** is not suspended by a reference v which is typed **may ?**. This is formally written:

$$\text{Must}(\mathsf{T}) \triangleq \forall u \in \mathsf{T}, (u{:}\mathbf{must\ !}M_\Sigma) \Rightarrow \forall v, u \rightarrowtail^* v, \neg(v{:}\mathbf{may\ ?}M_\Sigma)$$

The rules are based on the ones of Sect. 2, whereby *Com* is abstracted from the state structure.

4.1 Creation and Termination of a Port

The creation of a port means a new reference and its contract are added to $\tilde{U}$.

$$\text{CREAT}\frac{u : T \qquad B(\mathsf{P}, \mathsf{R}, \mathsf{T}) \rightarrow B'(\mathsf{P}[u \multimap \bot], \mathsf{R}', \mathsf{T}')}{(B(\mathsf{P}, \mathsf{R}, \mathsf{T}), \tilde{U}) \rightarrow (B'(\mathsf{P}[u \multimap \bot], \mathsf{R}', \mathsf{T}'), \tilde{U} \Leftarrow u : T)}\text{Must}(\mathsf{T}')$$

The termination of a port (i.e. its removal from the component) is allowed when the contract reaches **0** or **may !**.

$$\text{REMV}\frac{u:T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \to B'(\mathsf{P}\backslash u,\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \to (B'(\mathsf{P}\backslash u,\mathsf{R}',\mathsf{T}'),\tilde{U}\backslash u)}(T \equiv \mathbf{0} \text{ or } T \equiv \mathbf{may}\ \mathbf{!}\ M_\Sigma) \wedge \text{Must}(\mathsf{T}')$$

$$\text{REMV-ERR}\frac{u:T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \to B'(\mathsf{P}\backslash u,\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \to \mathit{Error}} T \not\equiv \mathbf{0} \text{ and } T \not\equiv \mathbf{may}\ \mathbf{!}\ M_\Sigma$$

4.2 Binding of a Partner Reference to a Port

When bound, the type of the partner reference has to be compatible with the port it is bound to:

$$\text{BIND}\frac{u:T \qquad u':T' \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \to B'(\mathsf{P}[u \multimap u'],\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \to (B'(\mathsf{P}[u \multimap u'],\mathsf{R},\mathsf{T}),\tilde{U})} \mathit{Comp}(T,T')$$

$$\text{BIND-ERR}\frac{u:T \qquad u':T' \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \to B'(\mathsf{P}[u \multimap u'],\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \to \mathit{Error}} \neg \mathit{Comp}(T,T')$$

The unbinding of a partner is allowed at any time (the only constraint is contained in the predicates of the rule CUNBIND).

4.3 Emitting and Consuming a Message

Message are emitted to a known partner reference. Modalities are expressed via compatibilities. A peer reference that is sent in the message must not be attached to a partner, and must be removed from R; this ensures the uniqueness of the peer role.

$$\text{SEND}\frac{u:T \equiv \mathit{mod}\ \mathbf{!}\ M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'!m_k} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \xrightarrow{u:u'!m_k} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{U} \Leftarrow u:T_k)} \blacktriangleleft$$

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

$$\blacktriangleleft \triangleq \tilde{v}_k : \tilde{T}'_k \wedge \big(\forall v \in \tilde{v}_k, \text{peer}(v) \Rightarrow (v \notin \mathit{CoDom}(\mathsf{P}) \wedge v \notin \mathsf{R}')\big) \wedge \text{Must}(\mathsf{T}') \wedge u' \notin \mathsf{P}$$

The first rule is the normal behavior of a component sending a message from port u; by the type constraints, the first action of the sent references must be **!**, and all the peer references must be removed (peer(v) means v is a reference of a peer) . The next rule stands for a message that is not allowed to be sent: in the case where sending is allowed, but the message is not in the list ($\neg m' : M_\Sigma$ stands for: $m' = M'(\tilde{v}')$ and $\forall k, M' \neq M_k \vee \neg \tilde{v}_k : \tilde{T}'_k$), and in the case where sending is not allowed. Note that the *****-construct is syntactically allowed only if $\rho = \mathbf{?}$.

When consuming a message, the modality constrains only the partner; however, the component has to be able to receive any message described by the corresponding typed interface.

$$\text{RECV}\ \frac{u : T \equiv mod\ \mathbf{?}\ M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\mathbf{?}m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \xrightarrow{u:u'\mathbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{U} \Leftarrow u : T_k, \Leftarrow \tilde{v}' : \tilde{T}'_k)}\blacktriangle$$

$$\text{RECV*}\ \frac{u : T \equiv mod\ \mathbf{?}M_\Sigma\mathbf{*} \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\mathbf{?}m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}|u'')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{U}) \xrightarrow{u:u'\mathbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}|u''),\tilde{U} \Leftarrow u'' : T_k, \Leftarrow \tilde{v}' : \tilde{T}'_k)}\blacktriangledown$$

$$\blacktriangle \triangleq len(\tilde{v}') = len(\tilde{T}'_k) \wedge \text{Must}(\mathsf{T}') \wedge u' \notin \mathsf{P}$$
$$\blacktriangledown \triangleq len(\tilde{v}') = len(\tilde{T}'_k) \wedge \text{Must}(\mathsf{T}') \wedge u' \notin \mathsf{P}$$

The first two rules describe the normal behavior when receiving a message (with correct number of arguments). We do not check the type of the arguments because if the message is sent, it was done 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. The only difference between the two rules is the spawning effect due to the *****-construct: the component creates a new port u'' to answer the request.

Rules for sending and receiving given here correspond to external interaction. For interactions between ports of the same component, different rules should be used, which involve collapsing two steps transition (a **!** and the corresponding **?**) into one transition. Those rules are not given here, for space limitations.

4.4 A must Is Not Honoured

This rule stands for all the error cases where a transition leads to a T' such that Must(T′) is false:

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

4.5 Component Honouring a Contract

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

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

5 Properties Guaranteed by the Compatibility Rules

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

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. It has the properties:

- the configuration is reference-closed: any partner reference designates a port of a component of the configuration,
- the only port bindings are peer to server bindings,
- and all the ports are active on independent threads.

$$\mathcal{A} = \{(B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1), \tilde{U}_1), \ldots, (B_n(\mathsf{P}_n, \mathsf{R}_n, \mathsf{T}_n), \tilde{U}_n), Com\}$$

$$\text{with } \begin{cases} \forall i, u: & u \in \mathsf{R}_i \Rightarrow \exists j \text{ such that } u \in \mathsf{P}_j \\ \forall u, v, i: & (u \multimap v) \in \mathsf{P}_i \Rightarrow \text{peer}(u) \wedge \text{server}(v) \\ \forall u \in \cup \mathsf{P}_i: & \mathsf{T}(u) = \rho^{\mathsf{a}} \end{cases}$$

An assembly, in its initial configuration, encompasses only client/server bindings. However, as it evolves, new peer-to-peer bindings may appear.

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

$$\mathcal{A} \text{ is sound iff } \begin{cases} \forall i: & B_i \models \tilde{U}_i, \\ \forall u{:}T_u, v{:}T_v, i \ : & (u \multimap v) \in \mathsf{P}_i \Rightarrow Comp(T_u, T_v) \end{cases}$$

5.2 Subject Reduction and Message Consumption Properties

The first property, P_{sr}, of a sound assembly states simply that soundness is maintained throughout the evolution. This kind of properties is called also subject reduction. P_{sr}, states that "a configuration of component never leads to *Error*":

$$P_{\text{sr}} \triangleq \forall \mathcal{C} : \mathcal{A} \rightarrow^* \mathcal{C}, \mathcal{C} \nrightarrow Error.$$

Theorem 1 (Subject reduction). *If $\mathcal{A}$ is sound, then $\mathcal{A} \vDash P_{\text{sr}}$*

Proof. The proof is 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. □

We define also P_{mc}, which stands for "all messages sent will eventually be consumed":

$$P_{\text{mc}} \triangleq \forall u, v, i, M : (u \multimap v) \in \mathsf{P}_i,$$
$$\mathcal{C} \xrightarrow{u:v!M} \mathcal{C}' \Rightarrow \exists \mathcal{C}'', \mathcal{C}''' \text{ such that } \mathcal{C}' \rightarrow^* \mathcal{C}'' \xrightarrow{v:u?M} \mathcal{C}'''$$

Corollary 1 (Message consumption). *If $\mathcal{A}$ is sound, then $\mathcal{A} \vDash P_{\text{mc}}$, modulo fairness.*

This corollary is a consequence of theorem 1 and the use of fifo queues. However, since the rules for consuming a message may be competing with others, we have to assume fairness in this competition.

5.3 External Deadlock Freeness

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

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

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.

But the general case is more complex and is formalized:

$$\begin{aligned}\text{Ext\_deadlock}(\mathcal{C}) \triangleq \exists (t_k)_{1..n} \in \text{threads}(\mathcal{C}), \exists (u_k)_{1..n}, (v_k)_{1..n} \text{ such that}\\ t_k = \cdots \rightarrowtail v_k \rightarrowtail \cdots \rightarrowtail u_k\mathbf{?}^{\mathbf{a}}\\ \wedge \big(\forall 1 \leqslant k < n, (v_{k+1} \multimap u_k)\big) \wedge (v_1 \multimap u_n)\\ P_{\text{edf}} \triangleq \forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C} \Rightarrow \neg\text{Ext\_deadlock}(\mathcal{C})\end{aligned}$$

Theorem 2 (External deadlock freeness). *If $\mathcal{A}$ is sound, then $\mathcal{A} \vDash P_{\text{edf}}$*

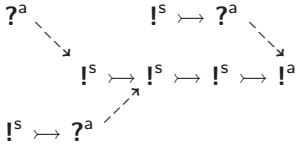
Proof of theorem 2 is tedious. Even if interfaces are mutually compatible, it is not straightforward that a deadlock will not arise between components (ports in a component may be suspended by another port, which leads to potential dependencies between threads added to dependencies between ports).

The deadlock-free 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 where the proof of the verification is not shown to be compositional.

Proof (External deadlock freeness). The sketch of the proof of theorem 2 is the following.

We define a new dependency relation between ports, namely, external dependency, denoted by $\dashrightarrow$, related to communications among remote ports. For example: $u\mathbf{?}^{\mathbf{a}} \dashrightarrow v\mathbf{!}^{\mathbf{s}}$

We use *dependency trees* to visualize the dependency relations. A dependency tree is an oriented tree in which nodes are of the form $u\rho^{\sigma}$, and links correspond to both dependency relations $\rightarrowtail$ and $\dashrightarrow$, directed from the leafs to the root (we do not consider idle references in those trees; it is straightforward from the rules on the components that those references will never have dependencies). Hence, the dependency trees correspond to the graphs representing the relation obtained by merging the two dependency relations. An example of such trees:



Dependency trees evolve along with the behavior of the components. Some evolutions are for example the merge of two trees, others may change the state of some node...

We then show by structural induction that cycle freeness in dependency trees is an invariant property. Since by definition this property is satisfied in the initial state ($\mathcal{A}$ starts with a set of independent threads), then external deadlock freeness is preserved throughout the derivations. □

5.4 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":

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

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 checked provided the source code of the component is available.

6 Conclusion and 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 anything different that what is described in its interfaces.

The verification of interface compatibility should instead be performed at the moment in which 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.

We have only applied our approach to some toy examples; we need to verify the usability of the approach in practice, especially with respect to the expressiveness of the interface language we have proposed. The conformance of the component model we have assumed with concrete notations (e.g. Java) should be studied: varying the component model to suite a concrete notation may actually affect the classes of properties that can be guaranteed. Also, we can 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

The authors from ENST have been partially supported by the RTNL ACCORD project and by the IST MIKADO project. The author from the University of Florence has been partially funded by the 5% SP4 project of the Italian Ministry of University and Research. The third author has been partially supported by a grant from ISTI of the Italian National Research Council.

Special thanks to Arnaud Bailly for his helpful advices.

References

[Bai02] A. Bailly. *Assume / Guarantee Contracts for Timed Mobile Objects*. PhD thesis, ENST, December 2002.

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

[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. Guaranteeing liveness in an object calculus through behavioral typing. In *Proc. of FORTE/PSTV'99*, Oct. 1999.