Composing Components with Shared Services in the Kmelia Model

Pascal André, Gilles Ardourel, and Christian Attiogbé

LINA - UMR CNRS 6241 - University of Nantes
F-44322 Nantes Cedex, France
{Pascal.Andre,Gilles.Ardourel,Christian.Attiogbe}@univ-nantes.fr

Abstract. The Kmelia abstract component model is extended to allow the description of component compositions with multipart interactions leading to simultaneous communications between more than two services. Shared services are defined to explicitly control multipart interactions. Accordingly the communication actions of Kmelia are extended. The formal definitions of the Kmelia model, the composition of components via their services and their analysis are revisited to integrate the extension of the model. An example illustrates the need and the usage of shared services.

Keywords: Component, Composition, Shared services, Multipart Communication.

1 Introduction

The Kmelia component model [3] was introduced as an abstract formal component model dedicated to the specification and development of correct components. The model is equipped with a language which is evolving together with the expressive power of the model. In [3] we have distinguished two semantics for the link between component services. Only one, *monadic semantics*, was treated in this previous article. The second one, *polyadic semantics*, was not treated. The hypothesis for the *monadic semantics* is: only one provided service may be associated to a required service; a component is both a component type and the unique instance of it; a required service may be linked to at most one provided service; only one instantiation of a service exists at any time.

In the current article we consider the *polyadic semantics*: a provided service may be linked with various required services (allowing broadcast communications); as an example, a chat system provides an interaction service for multiple clients. In the same way a required service may be linked to various provided services. We present the new features of our Kmelia model, the language aspects that support these features and how these improvements are integrated with the previous works on Kmelia.

Motivations. The modelling of various real life systems such as auction systems, chat systems, distributed brokers, etc requires the use of several components of the same type or several services with identical functionalities but coming

C. Pautasso and É. Tanter (Eds.): SC 2008, LNCS 4954, pp. 125–140, 2008.
© Springer-Verlag Berlin Heidelberg 2008

from different components. This leads to the need of interaction means to support the assembly and the composition w.r.t to the multiplicity of services that may be connected. The current Kmelia model and language provide a one to one service/component interaction even if several components participate in the assembly. This does not cover the kind of systems listed above.

Contribution. The contribution of this article is the improvement of the expressivity of the Kmelia component model with shared services, multipart interaction based on synchronous n-ary communications. We extend Kmelia to support multiple connections between services. Also, we explicitly distinguish between *component types* and *components (as elements)*, hence we may use several components of the same type in an assembly. Accordingly, the interaction between Kmelia services is updated.

The article is structured as follows. In Section 2 we give an overview of the Kmelia abstract model and we mention some new features introduced in this article. Section 3 is devoted to multiple links on the same service and the impact on the interaction between services. In Section 4 we deal with shared services and their impact on the assembly description. Section 5 shows an example of a component-based system with shared services and related interactions; Formal analysis issue is treated. The article is concluded in Section 6 where we discuss related works and give some perspectives to this work.

2 Overview of the Kmelia Model and New Features

2.1 Overview of the Kmelia Component Model

In [3,2] we have presented various aspects of our abstract component model called Kmelia. Here we recall the main elements of this component model and we build on them in order to improve the model according to the new communication features.

A Kmelia abstract component is a mathematical model of an open multiservice system that supports synchronous communication with its environment.

The main specification of a component [3] is preserved and referred to as the specification of a *component type*. The interface of a component is still made of required services and provided services. The core specification of a service is not changed. We recall the definition of a component; it stands now explicitly for a *component type*.

Component Type Specification. A component type (C) is a 8-tuple $\langle \mathcal{W}, Init, \mathcal{A}, \mathcal{N}, I, \mathcal{D}_S, \nu, \mathcal{C}_S \rangle$ with:

- $\mathcal{W} = \langle T, V, V_T, Inv \rangle$ the state space where T is a set of types, V a set of variables, $V_T \subseteq V \times T$ a set of typed variables, and Inv is the state invariant;
- $Init$ the initialisation of the V_T variables;
- $\mathcal{A}$ a finite set of elementary actions;
- $\mathcal{N}$ a finite set of service names;
- I the component interface which is the union of two disjoints finite sets: I_p the set of names of the provided services that are visible in the component environment and I_r the names of required services. We have $I \subseteq \mathcal{N}$.

- $\mathcal{D}_S$ is the set of service descriptions which is partitioned into the provided services ($\mathcal{D}_{S_p}$) and the required services ($\mathcal{D}_{S_r}$).
- $\nu : \mathcal{N} \rightarrow \mathcal{D}_S$ is the function that maps service names to service descriptions. Moreover there is a projection of the I partition on its image by ν: $n \in I_p \Rightarrow \nu(n) \in \mathcal{D}_{S_p} \wedge n \in I_r \Rightarrow \nu(n) \in \mathcal{D}_{S_r}$
- $\mathcal{C}_S$ is a constraint related to the services of the interface of $\mathcal{C}$ in order to control the usage of the services.

The behaviour of the component relies on the behaviours of its services. A service is *activated* by a call; It may activate other services during its evolution. Only one action of an activated service may be observed at time. Due to dependencies between services and interaction between components, the actions of several activated services may interleave or synchronise. The constraint $\mathcal{C}_S$ describes general conditions on the service usage: it can be an ordering of services or a predicate (safety properties, ...). Specific Kmelia provided services (called protocols) can *implement* a *Component Behaviour Protocol* in the sense of [9,13]. Kmelia allows the use of several protocols for the same component.

A service of a Kmelia component is defined with an interface and a behaviour. The interface is made of a signature, a pre-condition, a post-condition, a service dependency which gives the services on which the current one depends (*subs*: the subprovided services, *cals*: the service required from the caller, *reqs*: the services required from any component, *ints*: the internal services). The behaviour of a service is described with an extended labelled transition system. The labels may be either elementary actions (assignments, function call, ...) or communication actions which support the interaction between Kmelia services. Therefore a Kmelia service is not reduced to a single running stream from its start to its termination, when a service is called, it may have interaction with the caller or with other services.

A communication action is either a *service call/response* or a *message send/ receive*. The Kmelia syntax of a communication action (inspired by the Hoare's CSP) is: `channel(!|?|!!|??) message(param*)`. Therefore communication actions are matching pairs:

send message(!) - *receive message*(?),
call service(!!) - *wait service start*(??),
emit service result(!!) - *wait service result*(??)

We use the channel identifier `_CALLER` to denote the channel associated to a service for a call.

A Kmelia service s has callers (the services that call s) and callees (the required services that are called by s). When a service *reqServ* is required by a service s, the latter uses the channel named `_reqServ` to communicate with the service linked with *reqServ* in assemblies.

Assembly. An assembly is a set of components that are linked (composed) through their services; they interact via their activated services which communicate through the abstract channels that support the links established between the services. Graphically, a component is depicted as a box (See Fig. 1). On the

frontier of the component box, required services are depicted as empty small boxes included in the component box (like `rs1`). Provided services are depicted as empty small boxes outside the component box (like `ps1`).

2.2 New Features of the Kmelia Model

Component. A component is one element of a component type (see above Sect. 2.1). A component is referred to with a variable typed using the component type; for example `c1: CT` where `c1` is a variable and `CT` a component type. Several components of the same type will be denoted with `c[n]:CT` where `n` is a natural number. In the same way an assembly is one element of an **Assembly Type**.

Shared Service. Sharing is concerned with services and at low level with communication actions: several services may be involved in a communication, like in the broadcast messages.

A *shared provided service* is a provided service that can simultaneously interact with several services from other components. Therefore a subset I_{sp} of the interface I (where $I = I_p \cup I_r$) of a component constitutes the shared provided services of the component. Accordingly, $I_{sp} \subseteq I_p$.

As far as required services are concerned, we now allow that a (provided) service performs a simultaneous communication with the (provided) services that are linked to its required services. Therefore *required services may also be shared*. Sharing a required service forces the synchronisation of the linked services. A subset I_{sr} of the interface I of a component constitutes the shared required services of the component: $I_{sr} \subseteq I_r$.

Multipart communication. Within an assembly, a service may be linked to several other services, leading to a multipart communication between the involved services. For instance a provided service may send a message simultaneously to several callers or wait for a message coming from several callers. In the same way a service may simultaneously call all the services linked to its required service (*reqServ*) using the channel named `_reqServ`. A shared provided service may wait using `??` (resp. `?`) for a call (resp. a message emission) from several other services linked with it. Consequently the shared provided service sends a response (resp. a message) with `!!` (resp. `!`) to all its callers.

We also introduce the **role** concept to qualify some links and the related interactions.

In the following we give the details, the constraints of these kinds of interaction and the associated communication actions.

3 Shared Services: Impact on Service Interactions

The context here is multipart interaction between components via their services. The actions performed by the interacting services are interleaved but the services synchronise on communication actions. In our previous work, pairwise interactions are considered between components; only one-to-one linked services are

involved, and only one provided service may be linked to a required one. We consider now the cases where the interactions between several components are not restricted to one-to-one links between the services.

In the following we examine the various interaction cases with multiple services with respect to the expressivity of the Kmelia model. First we consider several provided services linked to one required, and one provided related to several required services. Then we generalise to a service interacting, even with synchronising communications, with several callers or several callees.

3.1 Linking Several Provided Services with one Required Service

A service *rs* required by a component may be fulfilled by one or several other services *ps* from one or several components (see Fig. 1). Each one of the *ps* services should be compatible with the requirement. Thus, at the specification level all the provided services linked with *rs* should be compatible with *rs*. The compatibility between services is already defined in [3].

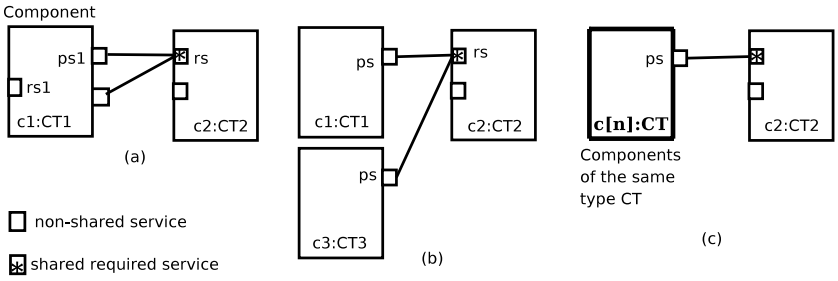

Fig. 1. Assemblies with Shared Required Services

If the provided services are from the same component (Fig. 1 (a)) or from different components (Fig. 1 (b)), the interaction may result in a synchronous multipart communication between, on the one hand, the provided services and on the other hand, the service that uses the required one (which should then be shared).

If the provided services are from components of the same component type (Fig. 1 (c)), the interaction may be specifically done with one component (provided that there is only one at time) or with several (or all) components if the caller service (associated to the required one) is designed to behave like this.

The expressive power of the Kmelia model had to be extended to cover the case depicted in Fig. 1 (a) and Fig. 1 (b) by considering simultaneous calls, message sending, call response, or message receiving from the provided services via the shared required service.

To handle the case represented by Fig. 1 (c), a service may call simultaneously *all* the provided services linked with it[1]; the called services may respond to

[1] There are other hypothesis that do not involve *all* the provided services; they are not considered here.

their common caller. Specific communication actions are needed to handle this kind of interaction which should be distinguished from those already existing in the Kmelia model (they are binary). From the semantic point of view, the called services evolve simultaneously and send their results back to the caller. We maintain here the use of a synchronous communication. The needed communication actions are introduced later in this section.

3.2 Linking one Provided Service with Several Required Services

In an assembly of components, several components may use one component and its provided services. In this case a given provided service may be linked with several required services (see Fig. 2). Practically, several services may call their shared provided either in exclusion with the other callers or simultaneously.

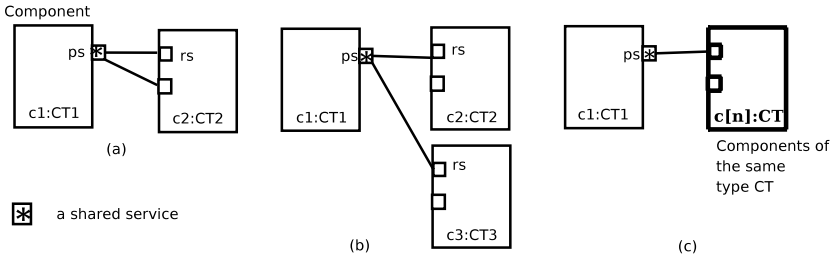

Fig. 2. Assemblies with Shared Provided Services

If we assume the exclusion between the running of the actions of the interacting services, the cases depicted with Fig. 2 (a) and Fig. 2 (b) will be correct interactions with respect to the current Kmelia model; but in this case the provided service (not shared) does not use multipart synchronous communication actions. However, sharing the provided service may force to multipart synchronous communication using a *wait for all*. Thus we consider simultaneous calls to the shared provided service (this also includes the case depicted in Fig. 2 (c), where several services of components of the same type are interacting with the same provided service), then the interaction between the provided service and the linked required services is not straightforward. We have a one-provided to n-required relationship. In this case the provided service should be shared: its communications are shared among the n callers which either belong to different components types (Fig. 2 (b)) or are components of the same type (Fig. 2 (c)). It does not matter to link a non-shared provided service to several callers, because each caller will interact separately in its call context. It is the provided service designer which should consider the ability to interact with several callers, otherwise there is no sharing.

To sum up we have to deal with the composition of a shared provided service with several services. The new feature to be treated is the composition of one

shared provided service with several required ones. From the caller side, there is no new requirement for the interaction. From the callee side, the interaction may be performed either with one specific caller, or with any caller, or with all the callers; it depends on the designer.

In the following we present the proposals to improve the communication actions of Kmelia in order to encompass the new communication needs.

3.3 Interaction with Shared Services

In this section we extend the communication actions of the Kmelia model to deal with the new kinds of interaction due to shared services.

Consider *sp* as a shared provided service; it may be called by several other services (the callers). The service *sp* may communicate with one specific caller, all the callers, or one among the callers. The interaction between the linked services is explicitly achieved from the shared provided service using different identifiers for the caller services. The communication actions (see Sect. 2.1) are now extended as multipart communication actions using a channel selector:

`channel[<selector>](!|?|!!|??)message(param*)`

The values of `<selector>` are: `ALL`, `i` and `:i`. For instance, `CALLER[ALL]` identifies all the callers; `CALLER[i]` identifies precisely the caller i where i is a natural number; `CALLER[:i]` will identify one of the callers, the identifier of which is then bound to i. These two last cases of communication are not detailed in this article. `ALL` stands for all the callers that are currently linked to a channel end. We introduce in Kmelia, additional communication actions (Tab. 1) to support the interaction between a shared provided service and its callers.

Table 1. Communication actions from a shared provided service to its callers

`CALLER[i]!msg(val)`	Emission of `msg(val)` to the caller `i`
`CALLER[ALL]!msg(val)`	Broadcast of `msg(val)` to all the callers
`CALLER[i]?msg(x: x_Type)`	Reception of a value from the caller `i`
`CALLER[:i]?msg(x: x_Type)`	Reception of a value from any caller `i`; the other received values are not taken into account.
`tab_x := CALLER[ALL]?msg(x: x_Type)`	Reception of values from all the callers, the received values are collected in a structure `tab_x` indexed with the identifiers i of the callers
`CALLER[i]!!subServ(val)`	Call of a sub-service of the caller `i`
`CALLER[ALL]!!subSerb(val)`	Broadcast of a sub-service call to the callers
`CALLER[i]??subServ(x: x_Type)`	Wait the return of a sub-service from the caller `i`
`CALLER[ALL]??subServ(x:x_Type)`	Wait the return of all sub-services from all the callers

In the current case, the communications are all synchronous. The case of `CALLER[:i]?msg(x: x_Type)` may be asynchronously treated but this is not the concern of the current article; only the synchronous semantics is considered.

As far as a shared required service *sr* is concerned, the services linked to *sr* are referred to using the default channel `_sr`. The extensions of communication actions are similar to those in the Tab. 1 replacing the channel `CALLER` with `_sr`.

3.4 Adding Roles to Interactions

When several services call a shared provided service, they may play different roles in the interaction. For example in a chat system where several members achieve a connection to a server and participate in discussions, one member may play the role of moderator. Distinguishing between several callers' roles gives more flexibility in the assemblies. Roles can be shared or not: for instance, moderator could be a non-shared role. A shared provided service can support multiple roles by suffixing the channel by the role identifier in communications that only concern a specific role. From the syntactic point of view we use the following form for the communication actions.

`channel:<RoleId>[<selector>](!|?|!!|??)message(param*)`

`<RoleId>` is a role identifier that qualifies the communication channel.

All the roles supported by a service *ps* should be fulfilled and every caller of the service should assume a role. Therefore considering a role of *ps* from a service *cs* can be done in two ways: either in the behaviour of *cs* by using a `<RoleId>` suffix in all its communication actions with *ps* or in the assembly by assigning `<RoleId>` to the link.

4 Shared Services and Component Assembly

4.1 Specification of Shared Services in Kmelia

It is the role of the specifier to qualify a provided or required service as *shared*. In the behaviour of a shared service, some transitions are labelled with the multipart communication actions described in Sect. 3.3. But a service may be declared to be *shared* without using the specific communication actions; it does not matter.

In the same way as it was done for the use of protocols in our model [2], we propose the use of qualifier. Therefore the interface of a shared provided service has the following forms (the same holds for shared required services):

```
shared provided serviceName(parameters)
       {... specification of the service ...}
```

or equivalently

```
provided serviceName(parameters)
       properties = {shared, ... }
       {... specification of the service ...}
```

An analysis of a service behaviour may lead to determine that it is shared or not. Formally we check that a provided service is effectively shared by checking the type of communication actions used in its behaviour. In the same way, we

formally check that a required service is effectively shared, by examining the communication labels of the services that use it.

A shared service (or subservice) may be used by a non-shared service (sub-service).

4.2 Composition: Component Assembly

The new communication means do not impact the definition of the assembly of components but they do impact the assembly correctness.

We recall that the composition of services is based on the links that support the interaction between the services. According to the use of component types, components and shared services, the following points are revisited:

- Assembly of components (explicitly the elements of given component types). An assembly as it is defined until now [3], is specified by considering components; therefore there is no changes to the assembly specification.
- Assembly of component types. It is an assembly defined from component types; it results in an assembly type and should be instantiated by specific components in place of the component types used in the assembly. A component type may appear more than once in an assembly.
- Component Composition. A component composition (via their services) is defined until now by considering the links and sublinks established via an assembly of components, by linking required and provided services. Now, we also permit the link of one provided service with several required ones. But, as *shared* provided/required services are provided/required services (inclusion property), the link and sublink definitions are still correct; they include shared provided services.
- Interaction and simultaneous evolving. The services in different components may evolve simultaneously with interleaving; an activated service may interact with another activated one which is linked to it. With the current improvement of the Kmelia model, a service may synchronise with several activated services from different components via the introduced communication actions (see Sect. 3.3).

In the following formal definitions, we use a set theory notation close to that of the Z or B languages where $X \leftrightarrow Y$ denotes the relation from X to Y (a set of pairs); dom and ran denote respectively the domain and the range of a relation; $a \mapsto b$ denotes the pair (a, b).

In the remainder let $\mathcal{C}$ be a set of C_k components with $k \in 1..n$ and $C_k = \langle\langle T_k, V_k, V_{T\,k}, Inv_k\rangle, Init_k, \mathcal{A}_k, \mathcal{N}_k, I_k, \mathcal{D}_{S\,k}, \nu_k, \mathcal{C}_{S\,k}\rangle$ as defined in Sect. 2. Let $\mathcal{N}$ be the set of service names of $\mathcal{C}$ ($\mathcal{N} = \bigcup_{k\in 1..n}\mathcal{N}_k$).

The formalisation of an assembly [3] remains mainly unchanged (for component and assembly) when we integrate the new communication actions of the Kmelia model. However we now distinguish explicitly components and component types; therefore we update here the involved parts of the existing formlisation.

Component Assembly Type. An assembly of components (recall from [3]) results in an Assembly Type; it is a composition of components described by a tuple $A = (\mathcal{C},\ links,\ subs)$ where $\mathcal{C}$ is a set of components, $links$ is a set of links between the component services and $subs$ is a relation from links to sublinks. It may be abstracted as $(\mathcal{C}_T,\ links,\ subs)$ by considering in $\mathcal{C}_T$ the types of the components C:

$$
\begin{array}{l}
links \subseteq Link \wedge \\
(1)\ (\forall (C_i, sn_1, C_j, sn_2) : links \bullet C_i \in \mathcal{C} \wedge C_j \in \mathcal{C} \wedge \\
\qquad ((sn_1 \in I_{p_i} \wedge sn_2 \in I_{rj}) \vee (sn_1 \in I_{ri} \wedge sn_2 \in I_{p_j}))) \\
subs : Link \leftrightarrow SubLink \wedge \\
(2)\ \operatorname{dom} subs = links \wedge \\
(3)\ (\forall ((C_i, sn_1, C_j, sn_2) \mapsto (C_k, sn_3, C_l, sn_4)) \in subs \bullet C_i = C_k \wedge C_j = C_l) \wedge \\
(4)\ (\forall (C_i, sn_1, C_j, sn_2) : \operatorname{ran} subs \bullet ((\nu_i(sn_1) \in \mathcal{D}_{Sp_i}) \text{ xor } (\nu_j(sn_2) \in \mathcal{D}_{Sp_j})))
\end{array}
$$

The linked components are the components of the assembly (1). The sublinks are related to links (2) that concern the same components (3). Provided services are linked to required services (1 and 4).

The links ($Link$) and sublinks ($SubLink$) between component services are specified as follows. The links are 4-tuple of component and service names with the following properties: (1) the service names are those of their owner components, (2) any component service is not linked to itself (not recursive).

$$
\begin{array}{ll}
BaseLink : I\!P\ (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N}) & \\
(1) & \forall (C_i, sn_1, C_j, sn_2) : BaseLink \bullet sn_1 \in \mathcal{N}_i \wedge sn_2 \in \mathcal{N}_j \\
(2) & \forall C_i : \mathcal{C},\ sn_1 : \mathcal{N}_i \bullet (C_i, sn_1, C_i, sn_1) \notin BaseLink
\end{array}
$$

A link connects two services of the interfaces of their owner components.

$$Link \subseteq BaseLink \wedge \forall (C_i, sn_1, C_j, sn_2) : Link \bullet sn_1 \in I_i \wedge sn_2 \in I_j$$

Assembly of Components. In the same way as a component is an element of a component type, a component assembly is one element of a component assembly type (viewed as a set of possible values of the defined assembly type and related properties, see above). A component assembly is referred to with a variable typed using the component assembly type; for example `ca: CAT` where `ca` is a variable and `CAT` a component assembly type.

Well-Formed Assembly Revisited. The well-formedness is modified as follows. A component *assembly* described by the triple $A = (\mathcal{C},\ links,\ subs)$ is a *well-formed component assembly* if the following properties hold:

- all the members of $\mathcal{C}$ are components;
- the services in the sublinks are not in the involved component interfaces, but they are in the dependencies of the involved services (w.r.t *sublinks*).

$$
\begin{array}{ll}
(5) & \forall (l, sl) \in subs \mid l = (C_i, sn_1, C_j, sn_2) \wedge sl = (C_k, sn_3, C_l, sn_4) \bullet \\
& \qquad ((sn_3, sn_1) \in depends_i{}^* \vee (sn_4, sn_2) \in depends_j{}^*)
\end{array}
$$

where $depends_i{}^*$ is the transitive closure of $depends_i$. The relation $depends_k$ between component services is defined as a part of the service dependency in a component C_k where $sm = \nu_k(m)$:

$$depends_k : \mathcal{N}_k \leftrightarrow \mathcal{N}_k$$
$$\forall (n, m) : depends_k \bullet (n \in cal_{sm}) \vee (n \in req_{sm}) \vee (n \in sub_{sm})$$

Practically a *link* establishes an implicit communication channel between the involved services. This channel is shared with the sub-services.

- when a service with a dependency (*subs*: the subprovided services, *cals*: the service required from the caller, *reqs*: the services required from any component, *ints*: the internal services) is shared, its dependencies *subs* are also shared.
- shared provided services are linked with one or several required services from one or several components. But non-shared provided services may also be linked with several required service. Therefore there is no specific assembly constraints. Correctness is checked w.r.t behaviours.
- only shared required services may be linked with several provided services. Let $linkedWithC(C, sn)$ be the set of links with the service sn of the component C; we have to check for the services linked to several other services (hence the use of card, the cardinal of a set).

$$linkedWithC(C, sn) = \{(C_i, sn_i, C_k, sn_k) \in link \mid C_i = C \wedge sn_i = sn\}$$

Let $sharedRequired(C)$ be a function that denotes the set of the shared required services of the component C. They are the services of Ir (the required services of C) which have the property `shared`.

$$(6) \quad \forall (C,\ sn) \mid C \in \mathcal{C} \wedge sn \in \mathcal{N} \wedge sn \in I_r \bullet \mathsf{card}(linkedWithC(C,\ sn)) > 1 \Rightarrow sn \in sharedRequired(C)$$

From the practical point of view, the parser-compilers of Kmelia specifications should be updated in order to raise some errors when the added well-formedness rules are not respected.

4.3 Composition: Composite Component

An encapsulation of a well-formed component assembly within a component type results in a composite component type. We have defined an operator named compose that builds a new component type by combining one or several components (see [3]). Inner component services are promoted at the interface of the composite component; the properties of the services are preserved by the promotion (for instance a shared service remains shared). In this paper, we do not emphasize other aspects of composition such as the access rules to inner components.

A well-formed assembly type cannot be used to build a composite component. It should be first instantiated with components. Informally, the instantiation of an assembly type $A_T = (\mathcal{C}_T,\ links,\ subs)$ consists in replacing each component type CT of $\mathcal{C}_T$ by a component with the type CT.

4.4 Revisiting Behavioural Compatibility Analysis

The behavioural compatibility of an assembly of components with multipart communication actions follows the general principle already formalised in the previous version of Kmelia [3], where we defined composability and behavioural compatibility analysis. The principle is: first, to consider a service s_i of a component C_i, one required service req of s_i, and one service s_j (of a component C_j) that is linked to req; the triple (s_i, req, s_j) constitutes the analysis context to check each service of C_i. Second, considering the labelled transitions B_i and B_j, that describe the behaviours of s_i and s_j, after checking the composability at service and component level, one should ensure $compatible(B_i, B_j)$ which is the interleaving of elementary actions and the matching of communication actions.

Now, the matching of communication actions is extended to multipart communications. To capture this aspect, we proceed as follows. The context of a service analysis, previously defined as a triple, is extended to: one service s_i, one required service req of s_i, and S_J the services linked to req. The third element of the triple may now be a set of services. Therefore checking the behavioural compatibility of (s_i, req, S_J), with B_i the behaviour of s_i and $\mathcal{B}_J$ the set of behaviours of the services s_j in S_J, results in:

i) checking (s_i, req, s_j) for each $s_j \in S_J$; that is denoted with:

$$\mathsf{compatible_gen}(s_i, S_J) \Leftrightarrow \forall s_j \in S_J \mid \ compatible(B_i, B_j)$$

with B_i the behaviour of s_i and B_j the behaviour of s_j

ii) checking *one-to-n matching* between s_i and S_J. They match if at each communication point we have the following matching conditions:

when s_i performs `_req[ALL]?msg(...)` each s_j in S_J performs `CALLER!msg(...)`;
when s_i performs `_req[ALL]!msg(...)` each s_j in S_J performs `CALLER?msg(...)`;
when s_i performs `_req[ALL]??srv(...)` each s_j in S_J performs `CALLER!!srv(...)`;
when s_i performs `_req[ALL]!!srv(...)` each s_j in S_J performs `CALLER??srv(...)`.

Formally this results in a synchronous communication between n communicating entities, where one of the entities synchronise with the other entities considered together. Recall the specification of the extended labelled transition system of a service s_i (from [3]): $s_i \mathrel{\widehat{=}} \langle S_{s_i}, L_{s_i}, \delta_{s_i}, \Phi_{s_i}, S_{0_{s_i}}, S_{F_{s_i}} \rangle$. The set $S_{0_{s_i}}$ contains the initial state of s_i; it may be used as the current state of s_i. Thus if $S_{0_{s_i}}$ is $\{cst_i\}$ then $((cst_i,\ \texttt{ll}),\ nst_i) \in \delta_{s_i}$ means that there is a transition labelled with `ll` from the current state cst_i to the state nst_i.

Using the previous matching conditions, we specify $\mathsf{one\text{-}to\text{-}n_matching}(s_i, S_J)$ as follows (only the first condition is expressed, the other ones are similar):

$$\frac{\begin{array}{c} s_i \mathrel{\widehat{=}} \langle S_{s_i}, L_{s_i}, \delta_{s_i}, \Phi_{s_i}, \{cst_i\}, S_{F_{s_i}} \rangle \ \wedge \ ((cst_i,\ \texttt{_req[ALL]?msg(...)}),\ nst_i) \in \delta_{s_i} \wedge \\ \forall s_j \in S_J \mid s_j \mathrel{\widehat{=}} \langle S_{s_j}, L_{s_j}, \delta_{s_j}, \Phi_{s_j}, \{cst_j\}, S_{F_{s_j}} \rangle \ \wedge \\ ((cst_j,\ \texttt{CALLER!msg(...)}),\ nst_j) \in \delta_{s_j} \end{array}}{\mathsf{one\text{-}to\text{-}n_matching}(s_i, S_J)}$$

Consequently behavioural compatibility is generalised to (s_i, req, S_J) with:

$$\frac{\mathsf{compatible_gen}(s_i, S_J) \wedge \mathsf{one\text{-}to\text{-}n_matching}(s_i, S_J)}{\mathsf{beh_compatible_gen}(s_i, S_J)}$$

From now on, the Kmelia model includes multipart interactions, synchronous synchronisation of several interacting services, and an up-to-date behavioural compatibility checking.

5 Experimentations and Formal Analysis

5.1 A Chat System with Shared Services

Consider a chat system made of a server component with the type CHAT_SRV and several client components with the type CHAT_CLT, see Fig. 3.

```
COMPONENT CHAT_SRV
INTERFACE
 provided: {connection,interaction}
 required: {}
SERVICES
provided  connection()
  {...}
shared    provided  interaction()
  //  sends 'news'
  //  receives 'msg', 'close'
  {...}
news ()
  {...}
END_SERVICES
```

```
COMPONENT CHAT_CLT
INTERFACE
 provided: {chat_session}
 required: {interaction}
SERVICES

required interaction()
  // receives 'news'
  // sends  'msg', 'close'
  {...}

provided chat_session()
  {...}
END_SERVICES
```

Fig. 3. The components CHAT_SRV and CHAT_CLT

In this system the server provides the services: `connection` to wait for connection from clients and `interaction` to exchange with the clients. Several clients may simultaneously interact with the server (the service `interaction` of the server is then shared). The actions performed during the interaction are: `msg` to receive/send messages from/to clients, `news` to broadcast messages to clients, etc. At any time a client may connect to the server, close the connection, send a message to the server, receive (and display) a message received from the server. Consider an assembly with one server (`srv1`) and three clients (`clt[3]`). The assembly is specified in Kmelia as depicted in Fig. 4. The behaviour of the main service (`chat_session`) of a chat client is depicted in Fig. 5.

```
COMPOSITION
{  srv1: CHAT_SERV
   clt[3]: CHAT_CLT  }
{  (p-r srv1.interaction, clt[3].interaction) }
```

Fig. 4. An assembly with one chat server and three clients

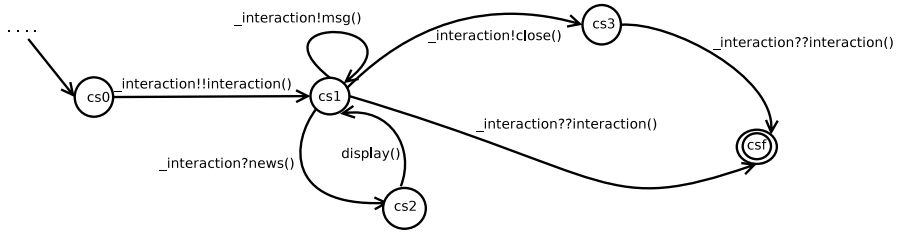

Fig. 5. A part of the behaviour of the `chat_session` service of the chat client

The behaviour of the service `interaction`, provided by the server component (CHAT_SRV), is depicted in Fig. 6.

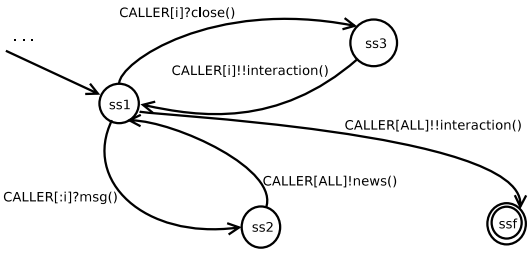

Fig. 6. A part of the behaviour of the `interaction` service of the chat server

5.2 Formal Analysis

Since the beginning we have designed the Kmelia model with the sake of pragmatism. For that purpose the COSTO toolbox [1] is being built.

The toolbox already enables us to parse Kmelia specifications, and to check behavioural compatibility using external tools such as LOTOS [10]. We defined bridges that translate Kmelia service specifications into LOTOS processes and we use the LOTOS/CADP[2][6] toolbox to check service properties including behavioural compatibility. Now, we have extended the expressive power of the Kmelia model; we have to provide or extend the tools to analyse Kmelia specifications. As far as behavioural compatibility is concerned we have to deal with multipart interactions involving synchronous n-ary communications.

N-ary communication supports are not generally provided by formal analysis frameworks. However LOTOS offers the negotiated multiway rendez-vous [10,7] that can be used for instance to model broadcast. We target this n-ary communication mechanism to partially analyse Kmelia multiway communications. Indeed several LOTOS processes may synchronise on the same gate G to exchange values. Thus the following communication actions from four LOTOS processes

[2] www.inrialpes.fr/vasy/

synchronise: `G!val`, `G?var1:T`, `G?var2:T` and `G?var3:T`. After the synchronisation the variables `var1`, `var2`, `var3` receive the value `val` sent by one of the processes on the gate `G`.

Processes may use negotiation to wait for a specific value; this is expressed with a guard (a predicate) following the wait action (`G?var:T [guard]`). The negotiated value is the one that satisfies the predicate of all the involved guards. It is also possible to synchronise with more than one emitted values (but they should be the same).

The new multipart communication actions may be performed using LOTOS processes. It is the case with `CALLER[ALL]!msg(...)` and `CALLER[ALL]!!srv(...)` which are broadcast. They are translated as a multiway communication between the processes associated to the caller services and the current process. The case of `CALLER[ALL]?msg(...)` is not straightforward; we have to collect all the values proposed by the environment; therefore we have to generate matching actions w.r.t the involved processes.

The current work in this direction is the extension of our translation modules of the COSTO tool in order to generate the LOTOS processes with the communication actions appropriate to the new features.

6 Discussion and Conclusion

Summary. In this paper we have presented some extensions to the Kmelia abstract component model: multipart interaction with synchronous communication; shared services; composition of component with shared services and multiway communication. The formal specification and analysis of the model are revisited accordingly.

Related works. In [12], a survey of component-based specification and architecturing languages is presented. The distinction between component types and their elements is widely used, it is the case for example with Wright[8], SOFA[13] and Fractal[4]. But some architecture description languages use a specific language to deal with type (Rapide[11] for instance). To our knowledge, component models do not support simultaneous interaction at the service level, but they allow multiple components connection (via connectors). SOFA and CCM[3] permits a connection from one to many components but no multipart communication between the services. Sharing is treated at component level in Fractal, in Kmelia we deal with communication and sharing at service level. More generally, the multiway communication among component services is not well-studied; one reason for that is the fact that several component models consider programming level instead of specification level. Component models based on the CSP process algebra may benefit from the synchronising n-ary rendez-vous to handle multipart synchronising interactions. Component models relying on programming levels (EJB, .NET) implicitely base synchonisation on execution threads. The current

[3] www.cca-forum.org

work engages a long-term investigation on this challenging subject through different abstraction levels.

Perspectives. Many aspects remain to deal with regarding sharing and the related properties, composition and correctness of component assemblies. We plan to investigate further the issues on multipart communication by considering the cases on selecting specific entities for a given communication. Another challenging point is the support for interoperability with other component models. The ideas under investigation are the structuring of the component interface (which should be more expressive) and the adaptation of the models with respect to the structuring of the information coming from other component model interfaces.

References

1. André, P., Ardourel, G., Attiogbé, C.: A Formal Analysis Toolbox for the Kmelia Component Model. In: Proceedings of ProVeCS'07 (TOOLS Europe), Technical Report. ETH Zurich, 567 (2007)
2. André, P., Ardourel, G., Attiogbé, C.: Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In: 6th International Symposium on Software Composition, SC 2007. LNCS, vol. 4829. Springer, Heidelberg (2007)
3. Attiogbé, C., André, P., Ardourel, G.: Checking Component Composability. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The Fractal Component Model and Its Support in Java. Software Practice and Experience 36(11-12) (2006)
5. Bruneton, E., Coupaye, T., Stefani, J.: Recursive and Dynamic Software Composition with Sharing. In: Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP 2002) (2002)
6. Fernandez, J.-C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP: A Protocol Validation and Verification Toolbox. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 437–440. Springer, Heidelberg (1996)
7. Garavel, H., Hermanns, H.: On Combining Functional Verification and Performance Evaluation Using CADP. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 410–429. Springer, Heidelberg (2002)
8. Garlan, D., Allen, R.: Formalizing Architectural Connection. In: Proceedings of the 16th ICSE, pp. 71–80. IEEE Computer Society Press, Los Alamitos (1994)
9. Giannakopoulou, D., Kramer, J., Cheung, S.-C.: Behaviour Analysis of Distributed Systems Using the Tracta Approach. ASE 6(1), 7–35 (1999)
10. ISO LOTOS. A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour. International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva (1988)
11. Luckham, D.C., et al.: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering 21(6), 336–355 (1995)
12. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
13. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, 2002. IEEE Transactions on SW Engineering, 28(9) (2002)