Checking Component Composability

Christian Attiogbé, Pascal André, Gilles Ardourel

LINA CNRS FRE 2729 - University of Nantes
F-44322 Nantes Cedex, France
(Christian.Attiogbe,Pascal.Andre,Gilles.Ardourel)@univ-nantes.fr

Abstract. Component-Based Software Engineering (CBSE) is one of the approaches to master the development of large scale software. In this setting, the verification concern is still a challenge. The current work addresses the composability of components and their services. A component model (Kmelia) is introduced; an associated formalism, simple but expressive is introduced; it describes the services as extended LTSs and their structuring as components. The composability of components is defined on the basis of the composability of services. To ensure the correctness of component composition, we check that an assembly is possible via the checking of the composability of the linked services, and their behavioural compatibility. In order to mechanize our approach, the services and the components are translated into the LOTOS formalism. Finally the LOTOS CADP toolbox is used to perform experiments.
Key words: Components, Services, Behavioural Interface Description, Composability, Behavioural Verification

1 Introduction

The rigorous development of large systems with methods that scale up and that are reusable in various projects is still a challenging research topic. Component-Based Software Engineering (CBSE) motivates a number of works on this topic [19, 15, 6, 12]. The component approach promotes the (re)use of components coming from third party developers to build new large systems. The success of the large scale development of component-based systems depends on the availability of: reliable components libraries, tools to search for components (in libraries), expressive languages of composition of the components and especially tools for checking the correct use of components.
The *motivation* for this work lies on the need of a sound basis for developing correct components, for studying composition and for implementing related tools. While many component approaches focus on the structural aspects of component composition, we insist on the functional (services) and dynamic (behaviour) aspects of the components because they are a main criteria for component reuse. In this perspective, related works deal with the behavioural compatibility for simplified abstract component models [6, 8, 5]. On the other hand there are mechanized approaches such as Tracta [10] or SOFA [17] but their component models are restricted. These works associate behaviour(s) to components and not to services. But this is a limitation since the services provide a finer description of the

component usage.
The *goal* of the work is to provide the designer of component-based systems with a high level component model and also with the methods to assist his/her use of the components. We are interested in building an experimental toolbox for component study and development.
The main *contribution* of this article is a simple formal model (named Kmelia) for modelling and composing components; it supports the verification of composability. We define composability of components by considering the links between their services and the behavioural compatibility of these services. Therefore, we get a hierarchical definition of composition and assemblies. In our work, a component is viewed and used through the *services* which constitute its interface. The use of services is central for the verification of composability when assembling components.
It is important to detect the defects which could lead to a faulty behaviour of the developed system early in the development. A bad interaction between a called service and the calling one may lead to a blocking of the whole system. To ensure a good level of correctness of the components and their assemblies, the formal verification of the service descriptions with respect to the desired properties of the component is necessary. Consequently, the specifications of components and their service behaviours should be abstract and formal. The use of an abstract formal model makes it possible to hide the implementation details of the components in order to have general reasoning techniques which are adaptable to various implementation environments.

The article is structured as follows. Section 2 presents the Kmelia model through the description of services and components. It is illustrated with an example of a bank Automatic Teller Machine system. Further details on Kmelia are given in [3]. The Section 3 introduces the service links and sublinks used to describe component assemblies and compositions. Section 4 is devoted to the composability of services and components. Behavioural compatibility between component services is also treated there. In the Section 5 we present the mechanization approach undertaken to support the Kmelia model. Experiments are done with LOTOS. Section 6 concludes with a discussion and the perspectives.

2 A Component Model Based on Services: Kmelia

In the Kmelia model, a component is characterised by: a name (the component identifier), a state (variables and an invariant predicate on them), an interface made of *services* and the description of the services. The *interface* specifies the component interactions with its environment [1, 15]. A Kmelia component interface is made of *provided services* and *required services*. A provided service offers a functionality, while a required service is the expression of the need of a functionality. This need is fulfilled when the component is combined with other components (in an *assembly*), one of them supplying the corresponding required service. Therefore, in Kmelia, component services interact with synchronous communication supported by message exchanges or service calls/responses via *communication channels*. Related works [17, 18, 16] associate dynamic behaviours

(or protocols) to components and services are atomic operations (*messages*). Unlike these approaches, we consider *services* as units of interaction and they are equipped with dynamic behaviours (service behaviours). This provides finer component descriptions where services are the main entities [2].

2.1 Service Specification

A *service* s of a component C is defined with an *interface* I_s and a (dynamic) *behaviour* $\mathcal{B}_s$: $\langle I_s, \mathcal{B}_s \rangle$. Usually a required service does not have the same level of detail as a provided service since a part of these details is already in the (provided) service that calls it.
The interface I_s of a service s is defined by a 5-tuple $\langle \sigma,\ P,\ Q,\ V_s, S_s \rangle$ where σ is the service signature (name, arguments, result), P is a precondition, Q is a postcondition, V_s is a set of local declarations and the *service dependency* S_s is a 4-tuple $S_s = \langle sub_s,\ cal_s,\ req_s,\ int_s \rangle$ of disjoint sets where sub_s (resp. cal_s, req_s, int_s) contains the provided services names (resp. the services required from the caller, the services required from any component, the internal services) in the scope of s. Using a required service r in cal_p of a service p (as opposed to a component interface) implies r to be provided by the component which calls p. Using a provided service p in the sub_r of a service r but not in the component interface, means that p is accessible only during an interaction with r.

The behaviour $\mathcal{B}_s$ of a service s is an *extended labelled transition system* (eLTS) defined by a 6-tuple $\langle S, L, \delta, \Phi, S_0, S_F \rangle$ with S the set of the states of s; L is the set of transition labels and δ is the transition relation ($\delta \in S \times L \to S$). S_0 is the initial state ($S_0 \in S$), S_F is the finite set of final states ($S_F \subseteq S$), Φ is a state annotation partial function ($\Phi \in S \to sub_s$). An eLTS is obtained when we allow nested states and transitions. This provides a means a flexible description with optional behaviours named *branching states* and also reduces the LTS size. A branching state is the one annotated with sub-service names (using the Φ function), which are (sub-)services of the component C that may be called when the evolution reaches this state (but the control returns to this state when the launched sub-service is terminated). Formally, the unfolding of (the branching states of) an eLTS results in an LTS.

Transitions: The elements $((ss, label), ts)$ of δ have the concrete Kmelia syntax `ss--label-->ts` where the labels are (possibly guarded) combinations of actions: `[guard] action*`. The actions may be *elementary actions* or *communication actions*. An elementary action (an assignment for example) does not involve other services; it does not use a communication channel. A communication action is either a *service call/response* or a message *communication*. Therefore communications are matching pairs: *send message(!)-receive message(?), call service(!!)-wait service start(??), emit service result(!!)-wait service result(??)*. The Kmelia syntax of a communication action (inspired by the Hoare's CSP) is: `channel(!|?|!!|??) message(param*)`.

Channels: A communication channel is established between the interacting services when assembling components. A channel defines a context for the communication actions. At the moment one writes a behaviour, one does not know

which components will communicate, but one has to know which channels will be used. A channel is usually named after the required service that represents the context. The placeholder keyword `CALLER` is a specific channel that stands for the channel open for a service call. From the point of view of a provided service p, `CALLER` is the channel that is open when p is called. From the point of view of the service that calls p, this channel is named after one of its required service, which is probably named p. The placeholder keyword `SELF` is a specific channel that stands for the channel opened for an internal service call. In this case, the required service is also the provided service.

2.2 Component Specification

A component (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.
- $\mathcal{D}_S$ is the set of service descriptions; it 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. The constraint $\mathcal{C}_S$ describes general conditions on the service usage: it may be an ordering of services or a predicate (temporal condition, mutual exclusion...). A specific service offered (like a *main*) as a single provided service may *implement* a *Component Behaviour Protocol* in the sense of [10, 17].

2.3 Component Assembly

Assembling Kmelia components consists in linking their pairwise services: required services may be linked to provided services. Formal details are given in the Sect. 3.3. Let consider two main semantics for the link operator: the *monadic* and the *polyadic semantics*. With the *monadic semantics*, 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 (no overloading); only one instantiation of a service exists at any time. The service evolutions are concurrent processes with shared component state. With the *polyadic semantics* a provided service may be linked with various required services (allowing broadcast communications); in the same way

a required service may be linked to various providers. As an example, a chat application provides services for multiple clients. Only the monadic semantic is considered in this article.

2.4 Illustration

We illustrate the model with a simplified real-world problem: a bank Automatic Teller Machine (ATM). Since the case is very common, the details are omitted here. Fig. 1 shows a simplified component assembly for the ATM, that includes four components: the central `ATM_CORE` which handles the ATM bank services, the `USER_INTERFACE` component which controls the user access, the `AAC` stands for the bank management and the `LOCAL_BANK` holds the local management access. The component usage is quite flexible: an assembly may be correct for the services whose dependency chains are fulfilled. The *USER_INTERFACE*

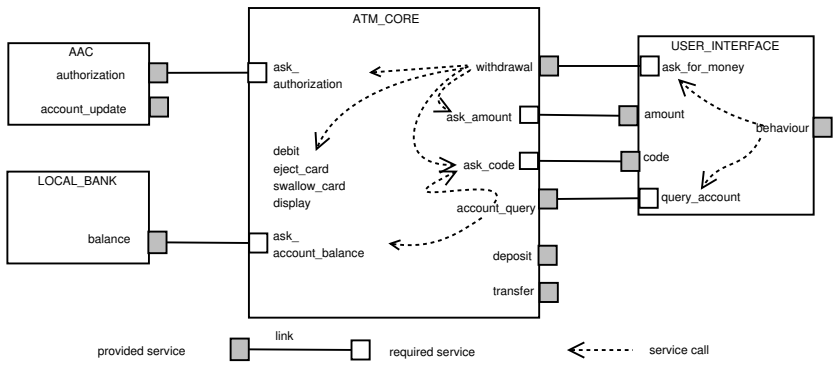

Fig. 1. Assembly for an ATM System

component offers the (provided) `code` service only in the interface of the `behaviour` service; it means that the *USER_INTERFACE* only gives its code during a withdrawal operation that it has initiated. In such a situation, `code` is a sub-service. The component services are detailed in the Fig. 2. Note that the *USER_INTERFACE* may also call a `withdrawal` service that does not require its `code`. In the following we focus on the `withdrawal` provided service which is linked to the required `ask_for_money` service, called by the `behaviour` service. This triple constitutes a context for a service verification (see Sect. 4.4). The links associated to `withdrawal` are:

```
(p-r ATM_CORE.withdrawal, USER_INTERFACE.ask_for_money
 //p-r stands for provided-required
 //sublinks
 (r-p ATM_CORE.ask_code, USER_INTERFACE.code)
 (r-p ATM_CORE.ask_amount, USER_INTERFACE.amount) )
```

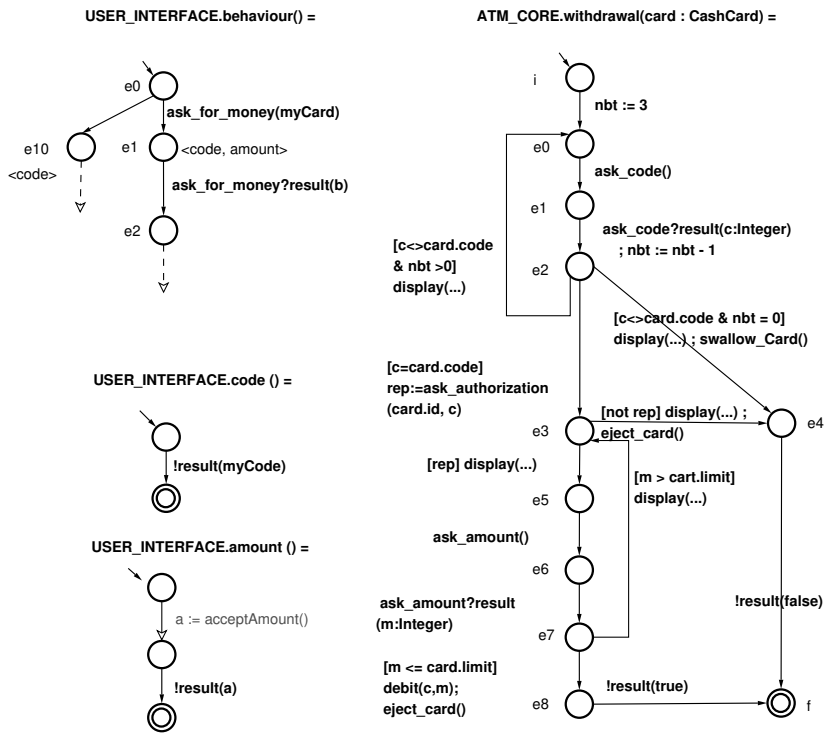

Fig. 2. LTS of the two main services (with the sub-services)

The withdrawal starts with an identification step: card insertion, password control, authentication by ACD/ATM controller (AAC). If the AAC accepts the transaction, the ATM asks for the amount of cash, otherwise the card is ejected and the withdrawal transaction ends. The user enters an amount which is compared with the current card policy limit. When the allowed amount is lower than the requested one or if the current ATM cash is not sufficient, the ATM asks again for the amount of cash. Otherwise the ATM asks the AAC to process the transaction, updates the card limit, gives the cash and prints a receipt when it is possible. In any case the withdrawal transaction ends after a card ejection. There are four elementary actions (*debit*, *eject_card*, *swallow_card*, *display*). The channels may be omitted and deduced either from the context or from default rules. This syntactic sugar is not currently implemented in our prototype.

The interaction description is made *flexible* by enabling the call of sub-services when the evolution reaches branching states. For example, the notation `e1 <code, amount>` expresses that the services `code` and `amount` may be called in the `e1` (branching) state. Thus the `ask_for_money` service may operate with

any withdrawal protocol (whatever the order for amount and code). The angle brackets are the syntactic counterpart of the Φ function.

3 Component Assembly and Composition

In the Kmelia model, the component assembly and composition are based on various types of *links between services*. For instance we have a *sublink* when a hidden service (not in the interface of the components) is called in the scope of a provided service. In an assembly, required services are linked to provided services. A composition is an assembly where some unlinked services are promoted to the composite level.

In this section, we provide the formal background for component assembly and composition. We use a set theory notation close to that of Z or B where $X \leftrightarrow Y$ denotes the relation from X to Y (a set of pairs), id denotes the identity relation; 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
$C_k = \langle\langle T_k, V_k, V_{Tk}, Inv_k\rangle, Init_k, \mathcal{A}_k, \mathcal{N}_k, I_k, \mathcal{D}_{Sk}, \nu_k, \mathcal{C}_{Sk}\rangle$ as defined in Sect. 2.
Let $\mathcal{N}$ be the set of service names of $\mathcal{C}$ ($\mathcal{N} = \bigcup_{k \in 1..n} \mathcal{N}_k$).

3.1 Dependencies between Component Services

Let $depends_k$ be a relation between component services defined as a part of the service dependency in a component C_k where $sm = \nu_k(m)$:

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

3.2 Links and Sublinks between Component Services

Basically, 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).

$$BaseLink : I\!P\ (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N})$$
$$(1) \quad \forall (C_i, n_1, C_j, n_2) : BaseLink \bullet n_1 \in \mathcal{N}_i \wedge n_2 \in \mathcal{N}_j$$
$$(2) \quad \forall C_i : \mathcal{C},\ n_1 : \mathcal{N}_i \bullet (C_i, n_1, C_i, n_1) \notin BaseLink$$

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

$$Link \subseteq BaseLink \wedge \forall (C_i, n_1, C_j, n_2) : Link \bullet n_1 \in I_i \wedge n_2 \in I_j$$

A sublink is a base link between two services such that one of them at least is hidden. For instance we have a sublink when a hidden service is called in the scope of a provided service.

$$SubLink \subseteq BaseLink \wedge \forall (C_i, n_1, C_j, n_2) : SubLink \bullet n_1 \notin I_i \vee n_2 \notin I_j$$

The *sublink* makes explicit the relation between the service dependencies declared in the interfaces of the services involved in a *Link*. In the following these relationships are constrained in order to define a specific component assembly and component composition.

3.3 Component Assembly

A component assembly is a triple $A = (\mathcal{C},\ links,\ subs)$ where $\mathcal{C}$ is a set of components, $links$ is a set of links between the services of $\mathcal{C}$ and $subs$ is a relation from links to sublinks.

$$
\begin{array}{ll}
& links \subseteq Link \land \\
(1) & (\forall (C_i, n_1, C_j, n_2) : links \bullet C_i \in \mathcal{C} \land C_j \in \mathcal{C} \land \\
& \quad ((n_1 \in I_{p_i} \land n_2 \in I_{r_j}) \lor (n_1 \in I_{r_i} \land n_2 \in I_{p_j}))) \\
& subs : Link \leftrightarrow SubLink \land \\
(2) & \operatorname{dom} subs = links \land \\
(3) & (\forall ((C_i, n_1, C_j, n_2) \mapsto (C_k, n_3, C_l, n_4)) \in subs \bullet C_i = C_k \land C_j = C_l) \land \\
(4) & (\forall (C_i, n_1, C_j, n_2) : \operatorname{ran} subs \bullet ((\nu_i(n_1) \in \mathcal{D}_{S_{p_i}}) \text{ xor } (\nu_j(n_2) \in \mathcal{D}_{S_{p_j}})))
\end{array}
$$

The components of the links 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 triple A is a *well-formed component assembly* if the following property holds: 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*).

$$
\begin{array}{ll}
(5) & \forall (l, sl) \in subs \mid l = (C_i, n_1, C_j, n_2) \land sl = (C_k, n_3, C_l, n_4) \bullet \\
& \quad ((n_3, n_1) \in depends_i{}^* \lor (n_4, n_2) \in depends_j{}^*)
\end{array}
$$

where $depends_i{}^*$ is the transitive closure of $depends_i$.

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

3.4 Component Composition

A *composition* is a well-formed component assembly which is encapsulated within a component. We define an operator named compose that builds a new component by combining one or several components.
The parameters of the compose operator are:

- an outer component oC (the composite) together with its interface, new services and services of its constituents;
- a well-formed assembly $A = (\mathcal{C},\ links,\ subs)$ (see section 3.3);
- the desired *promotions*, that are set of links between the services of oC and those of $C_k \in \mathcal{C}$.

The promotion is a relation between a service of the composite oC and an unlinked service of the components in A, that preserves existing sublinks; such

promoted service becomes usable at the composite level.

$$
\begin{array}{ll}
& promotions \subseteq BaseLink \wedge \\
& \quad (\forall (C_i, n_1, C_j, n_2) : promotions \bullet \\
(1) & \quad (C_i = oC) \wedge (C_j \in \mathcal{C}) \wedge \\
(2) & \quad ((\nu_{oC}(n_1) \in \mathcal{D}_{Sp_{oC}} \wedge n_2 \in I_{p_j}) \vee (\nu_{oC}(n_1) \in \mathcal{D}_{Sr_{oC}} \wedge n_2 \in I_{r_j})))
\end{array}
$$

The resulting component is an enhancement of oC: it contains every provided and required services of oC and provides/requires the promoted services from other components in $\mathcal{C}$ (using *promotions*). Here the sub-services of the promoted services are also promoted.

From the methodological point of view, the composition operator may be used to refine an abstract component with a component assembly; it may also be used to structure simple components or to provide a more restrictive interface of existing components.

4 Formal Verification of Components and Assemblies

Formal verification of components is performed according to various aspects. In the Section 4.1, we overview the main issues of component formal verification so as to situate the composability of components. Thereafter we focus on one specific aspect: the verification of the correct interaction between components. Indeed, a part of the service composability lies on the behavioural compatibility of the services: a correct service interaction is a guarantee for the composition of components. In the following, both static aspect and dynamic aspect of the verification are considered to check composability.

4.1 Formal Analysis of Components

The *safety* and *liveness* verifications apply to software components; but they should be adapted to component features. The *behavioural compatibility* between components is related to both safety and liveness. It is a widely studied topic [21, 8, 4, 7]. Behavioural introspection (discovering the component behaviour) is one way to deal with behavioural compatibility; but one has to prove compatibility. Checking behavioural compatibility often relies on checking the behaviour of a (component based) system through the construction of a finite state automaton. However the state explosion limitation is a flaw of this approach [8, 4]. More generally, the following properties should be considered for verification.

- *(Static) Interoperability properties*: it is the compatibility of signatures and interfaces (naming and typing); do a component gives enough information about its interface(s) in order to be (re)usable by other components;
- *Architectural properties*: that means the availability of the required components, the availability of the needed services, the correctness of the links between interfaces of components (providers and callers);
- *Behavioural compatibility*: it is about the correct interaction between two or more components which are combined. Several points need to be considered: various kinds of interaction, synchronous or not, atomic actions or non atomic ones.

– *Correctness of functional properties*: do the components do what they should do? These properties may be independently checked on the components which are used and also on the composition of the components;
– *Flexibility of maintenance (modifiability, evolution)*: that means the components should be simply updated on needs, without affecting drastically the third party components which use them. The update of a component includes, the modification of the implementation of its service(s), the removing/adding of a service, etc.
– *Heterogeneity*: within the CBSE approach, the components coming from various providers may be composable to develop large systems. This is a challenging concern because the components may have different models.
– *Compositionality*: the properties of a global system should be deduced from the properties of the composed components; it is an important concern.

The first three categories of properties are related to composability.

4.2 Composability

We define composability at different related levels: service level and component level.

Definition 1 (Service Composability).
A provided service $sp_{C_i} = \langle\langle \sigma_p, P_p, Q_p, V_{sp}, S_{sp}\rangle, \mathcal{B}_{sp}\rangle$ *of a component* C_i *and a required service* $sr_{C_j} = \langle\langle \sigma_r, P_r, Q_r, V_{sr}, S_{sr}\rangle, \mathcal{B}_{sr}\rangle$ *of a component* C_j *are s-composable (noted s-composable*(sp_{C_i}, sr_{C_j})*) when* sr_{C_j} *is required in the behaviour* $\mathcal{B}_s$ *of a service* s *of* C_j *if:*

1. *the interfaces of* sp_{C_i} *and* sr_{C_j} *are compatible; that is,*
   (a) *their signatures are matching (no type conflict:* σ_p *and* σ_r *are identical),*
   (b) *the assertions (pre/post) are consistent (*$post(sp_{C_i}) \sim post(sr_{C_j})$*) and*
   (c) *their mutually dependent services* S_{sp}, S_{sr} *(see service dependencies in Sect. 2.1) are not conflicting: the inner required-provided relationship is preserved: that means they involve a well-formed assembly(see Sect. 3.4).*
2. *the behaviour* $\mathcal{B}_{sp}$ *of* sp_{C_i} *and* $\mathcal{B}_s$ *of* s *are compatible: compatible*$(\mathcal{B}_{sp}, \mathcal{B}_s)$*; that is, their eLTSs are matching; either they evolve independently or they perform complementary communication actions until a termination without a deadlock.*

The conditions 1.c. and 2. are checked in the context of each service s that calls sr_{C_j}. $\mathcal{B}_{sr}$ is nul since the required sr_{C_j} does not have a behaviour. The compatibility of behaviours is dealt with in more details in the following.

Definition 2 (Component Composability).
Two components C_i *and* C_j *are c-composable according to a set of service pairs* ss*, if all the pairs* (s_i, s_j) *of* ss *are composable:*
$c\text{-}composable(C_i, C_j, ss) \Leftrightarrow \forall (s_i, s_j) \in ss \bullet s\text{-}composable(s_i, s_j)$

Proposition 1 (Assembly 'Checkability'). *When two components* C_i *and* C_j *are c-composable wrt to a list of services* ss*, then* C_i *and* C_j *can be linked in a well-formed assembly via* ss*. This generalises to several components.*

Accordingly Kmelia component assemblies and compositions may be formally checked for correctness.

4.3 Checking Composability: Static Analysis

The interface of a component contains the sets of provided and required services (with the naming and typing informations); additionally, informations on required or called sub-services are attached to the interface. In a similar way, these informations are available for the service descriptions. Accordingly, the static analysis of the interface of a component is achieved using:

i) simple correspondence checking algorithms and possibly standard typing algorithms;
ii) a deep investigation on the availability of required or called sub-services.

The definitions given above are used to perform this static level analysis. At this stage, some incompatibilities may be detected. We cover by the way a main part of (static) interoperability properties and architectural properties.

4.4 Checking Composability: Behavioural Compatibility Analysis

At this stage, we assume that a verification of the *static and architectural properties* is already performed for a given assembly. This implies that each service of the components is completely and correctly described. Now, the main concern is to check that a given component interacts correctly with others (which may be provided by a third party developer) over the links. Remind that each service is described with an eLTS where the transitions are labelled with guarded elementary actions and communication actions (see Sect. 2.1).

The component interacts correctly with its environment if its services are composable with the other services. We consider only one caller service and one called service at time. We check that *Bp* a given eLTS matches with *Br* a second eLTS: *compatible(Bp, Br)*. A complete interaction between the services of several components results in a pairwise local analysis between the LTS of a caller and that of the called service. The eLTSs are unfolded to obtain LTSs. Therefore, two services interact until a terminal state if the labels of their associated LTSs are in correspondence according to a set of rules that define *compatible*. They are based on the labels of the transitions going from a current state to the following states (output transitions). The rules indicate the correct evolutions according to the current states of two involved services: from a current state considered in each LTS, we explore the labels on the output transitions. In the case of elementary actions on the labels, each LTS evolves independently, their current states are updated. In the case of communication actions on the labels, the transitions match if for the considered services (hence the appropriate channels), we have the matching pairs: *send(!)-receive(?), call service(!!)-wait service start(??), emit service result(!!)-wait service result(??)*. In this case each LTS evolves in its next state. If the labels do not match, an incompatibility or a deadlock is detected. After a final state of a called service, the caller may continue with independent

transitions or with transitions that imply other (sub-)services. When the final states are reached without deadlock, the services are compatible.

In the following we focus on a practical verification of the *behavioural compatibility* aspect, that (re)uses an existing verification toolbox.

5 Formal Verification with LOTOS/CADP

We use LOTOS [14] and its associated CADP [9] toolbox to experiment on the composability checking. We encode the Kmelia components into LOTOS processes which are the input of the CADP tools. In order to exploit the CADP tools, the behavioural compatibility is based on communication between processes.

5.1 LOTOS

LOTOS [14] is an ISO standard formal specification language. It is initially designed for the specification of network interconnection (OSI) but is also suitable for concurrent and distributed systems. LOTOS extends the process algebra CCS and CSP and integrates (algebraic) abstract data types. A LOTOS specification is structured with process behaviours. It has the main behaviour description operators of the basic process algebra CCS and CSP. LOTOS uses the "!" and "?" operators of CSP which denote respectively emission and reception. The salient features of LOTOS are: the powerful multi-way synchronisation; the use of communication channels called *gates*; the synchronous interaction of processes; the use of algebraic data types to model data part of systems; the availability of the CADP toolbox [9].

A process is the description in the time, of the observational behaviour of a given system. The description is given as the non-deterministic combination of the sequence of events feasible by the system. The set of events of a behaviour is called the *alphabet* of the process. In a process specification, a sequence of events is denoted with ";". The choice between alternative behaviours B and C is described with `B [] C`. The notation `[Bterm] -> B` describes a process behaviour `B` guarded with a boolean term `Bterm`. The inaction is denoted with `stop`. A successful termination is denoted with `exit`. The sequential composition of behaviours `B` and `C` is described with `B >> C`.

Three parallel composition operators are used to compose processes: `|||` is used for the interleaving behaviour of the composed processes; `||` is used for the strict (on all the events) synchronisation of the involved processes; `|[L]|` where `L` is a synchronisation list (of events) is used to synchronise the processes on the events within the list L; when L is empty this results on a interleaving. The use of `L` forces the related processes to perform matching communication actions. Both synchronous and asynchronous communications may be described in LOTOS.

The ISO LOTOS has an operational semantics in terms of labelled transitions systems. The semantic rules define the behaviour of the LOTOS processes and their communication. As far as the data part is concerned, algebraic term rewriting is considered to evaluate data terms and each variable may be instantiated by the values corresponding to its type.

5.2 Translating the Services into LOTOS Processes

Our working hypotheses are the followings. To deal with the communication, each service has a *default channel* made by prefixing the service name with the keyword "`Chan_`". Thus, `Chan_serv` denotes the default channel of a service named *serv*. This channel is used as an alphabet element of the process corresponding to the service. In the same way, the channels associated to the services with which a service *serv* communicates (service calls appearing in the behaviour) are listed in the alphabet. We treat the activation of a service with a communication (to enter the initial state of the called service). A process corresponding to a service waits for a call. The caller service sends a call. Initially each service (the associated process) waits for a communication using its default channel. A caller service calls a service by sending a message (with the called name as parameter) on the default channel of the called service. The parameters are also sent using the default channel of the called process.

Translation Principle and Result

Remind that the behaviour of a component service is a transition system where each label is a combination of actions which may be elementary actions, or communication actions. From each state of a service there are one or several (outgoing) transitions going to other states.

LOTOS processes are basically state machines. Therefore the transition system which describes a service is described with one or several LOTOS processes; one main process is associated to the service and one or several subprocesses are used to described the former one. Basically, each state is translated into a process. The behaviour of the latter describes the transitions which are attached to the corresponding state.

The translation procedure is performed as follows: each service state is examined; each outgoing transition of the state corresponds to a LOTOS action followed by the translation of the reached state. The used channels, the communication actions and the elementary actions are collected to form the current process alphabet. According to these working hypotheses, we define a semantic encoding (namely *LotosEncoding*) of the service specifications. The encoding into LOTOS of service specifications is inductively performed by considering: service interface without formal parameters; service interface with formal parameters; service states (initial, final, intermediary and branching) and the transitions related to each service state.

During the translation process, the data type spaces are reduced[1] to avoid the state explosion problem: we use enumerated or byte types. For each service *ServName*, we define a LOTOS data type. It has a constructor which is named according to the service; this permits the call of the service. Besides, all the messages which are sent to the default channel associated to a service are used as constructors of the data type associated to this service. Enumerated data are translated with constructors of abstract data types. The expressions used within

[1] Model checking tools consider all the possible values of a type.

actions are not evaluated; they are translated by simple actions in the LOTOS process. The guards are not evaluated; each guard is encoded by an action.

5.3 Using CADP to Check the Behavioural Compatibility

The behavioural compatibility checking is based on LOTOS processes communication. We use the `|[L]|` composition operator. A compatibility checking involves a pair of services: the caller service and the called one; for example `behaviour` and `withdrawal` in our case (see Fig. 2). The `withdrawal` service is required by `behaviour` via the name `ask_for_money`. A renaming of `withdrawal` with `ask_for_money` is performed. These two services (the caller and the called) are translated into LOTOS processes (say `Lbehaviour` and `Lask_for_money`); each process has its alphabet (`alphabet` in the following); the processes are then composed using the `|[L]|` operator to get a resulting process called `Res` in the following. `L` is instantiated with the list of channels and actions used for the communication between both services as illustrated above.

```
Res =  Lbehaviour[alphabet]
    |[chan_behaviour, chan_ask_code, chan_ask_amount, ...]|
       Lask_for_money [alphabet]
```

Consequently, the services are compatible if the obtained `Res` process has no conflict according to the composition operator.
As far as the running example is concerned, we check that `USER_INTERFACE` and `ATM_CORE` are composable according to the services (`ask_for_money, withdrawal`): the interface checking is easy. The behaviours of `ask_for_money` and `withdrawal` are compatible.

To make it easy the experimentation of our component model, we implement an analyser (using Antlr[2] and Java) of component specifications. A prototype (named `kml2lotos`) to translate the component services into LOTOS is also developed using Java.

Given an input component specification (in Kmelia), the analyser parses the specification and generates the corresponding internal structure. The latter is read by the `kml2lotos` prototype; it generates communicating LOTOS processes which are used as input to the CADP toolbox. In the ATM case study (see Section 2), the experiment deals with an assembly of components. Specific services (a caller with a called one, branching node with the sub-services) are checked. The CADP functionalities raise failures when there are lack of channels, wrong channels, incompatible types, blocking or incompatible behaviours.

The experiment using CADP helps us to discover specification errors; for example when a wrong communication channel is used. When the errors are recovered and the communications are fine, the CADP `caesar` utility generates the (execution) graph corresponding to the system. The graph is very large in the case of brute translation; but when we erase independent alphabet actions and minimise the generated graph, we get a graph with less than hundred states.

[2] www.antlr.org

Stepwise simulation (using CADP `executor` utility) is performed to analyse the evolution of the system.

6 Discussion and Perspectives

We have presented a model where a component provides several behaviours via services. This flexibility offered to the user results in a non trivial formalisation of the model and its composability. A formal model is built to serve reasoning purpose and the composability is defined. Composable components may be used to build component assemblies or compositions. Some experiments are performed with the LOTOS CADP toolbox. A prototype toolbox (COSTO: COmponent Study TOolbox) is under development to support our experiments; it already integrates some modules: a Kmelia analyser, an architectural correctness checker, a translator to generate LOTOS processes from the component specifications. We also have a translator to MEC.

Compared to related works [4, 13], our approach works at the abstract specification level, it offers a more flexible formalism than the ones proposed by [21, 4] for the description of interacting services. We adopt a pairwise verification approach that avoids state explosion like in [4]. From the practical point of view, our proposal follows the mechanized approaches like Tracta [10] or SOFA [17]. The latter already provides many analysis tools; but we have a different component model that needs deep investigation before tool reuse and development. However we can build on the experiences gained with these works. Most of the approaches that integrate behavioural specifications to components [17, 16, 18] work at a protocol (or component) level while our approach is mainly based on the services, the protocol level is handled by a constraint in our model. Moreover, their communication actions refer only to messages and not to services (no service call or result). The non-regular protocols of [18] may be represented in Kmelia using guards and nested states, but using algebraic grammar provides a more efficient solution for the given applications. The work of [16] addresses assemblies and implementation issues in Java but does not deal with composition.

Many exciting investigations remain to be done. Whatever the component model, the compositionality is still a challenge [20]. The perspectives of this work are: to reinforce the correctness properties of component with supplementary study of correctness of components and services with regard to their environment; to extend the COSTO (COmponent Study TOolbox) prototype under development so as to cover more mechanized analysis concerns.

References

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
2. P. André, G. Ardourel, and C. Attiogbé. Behavioural Verification of Service Composition. In *ICSOC/Workshop on Engineering Service Compositions*, WESC'05.
3. P. André, G. Ardourel, C. Attiogbé, H. Habrias, and C. Stoquer. A Service-Based Component Model: Formalism, Analysis and Mechanization. Technical Report RR05.08, LINA, December 2005.

4. P. C. Attie and D. H. Lorenz. Correctness of Model-based Component Composition without State Explosion. In *ECOOP 2003 Workshop on Correctness of Model-based Software Composition*, 2003.
5. P. C. Attie and D. H. Lorenz. Establishing Behavioral Compatibility of Software Components without State Explosion. Technical Report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, 2003.
6. K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, New York, NY, 2000.
7. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
8. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.
9. J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. of the 8th Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 437–440. Springer Verlag, 1996.
10. D. Giannakopoulou, J. Kramer, and S.C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *ASE*, 6(1):7–35, 1999.
11. T. Gschwind, U. Aßmann, and O. Nierstrasz, editors. *Software Composition, 4th Int. Workshop, SC 2005, Edinburgh, UK*, volume 3628 of *Lecture Notes in Computer Science*. Springer, 2005.
12. G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors. *Component-Based Software Engineering, 8th International Symposium, CBSE 2005, USA, May, 2005*, volume 3489 of *LNCS*. Springer, 2005.
13. P. Inverardi, A. L. Wolf, and D. Yankelevich. Static Checking of System Behaviors using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, 2000.
14. 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.
15. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, january 2000.
16. S. Pavel, J. Noyé, P. Poizat, and J.C. Royer. A Java Implementation of a Component Model with Explicit Symbolic Protocols. In Gschwind et al. [11], pages 115–124.
17. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on SW Engineering*, 28(9), 2002.
18. M. Südholt. A Model of Components with Non-regular Protocols. In Gschwind et al. [11], pages 99–113.
19. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Publishing Company, 1997.
20. F. Xie and J. C. Browne. Verified Systems by Composition from Verified Components. In *ESEC/FSE-11: Proc. of the 9th European software engineering conference*, pages 277–286, New York, NY, USA, 2003. ACM Press.
21. D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.