# Behavioural Verification of Service Composition

Pascal André, Gilles Ardourel, Christian Attiogbé

LINA - CNRS FRE 2729, University of Nantes
Gilles.Ardourel@univ-nantes.fr

**Abstract.** In this work, we present a model for the definition, the composition and the verification of services. In this model, the services are bound to components. The composition of components results in the interaction of services. An inconsistency in this interaction foretells the failure of the service composition. The contributions of this work are: *i)* a formalism that allows a flexibility in the description and the composition of services: optional behaviours, optional sub-services and renaming; *ii)* service interfaces enriched with behavioural informations that contribute to the early detection of service incompatibilities; *iii)* the verification of the behavioural compatibility of service composition. The verification process is based on LTS techniques; we reuse existing verification tools.

## 1 Introduction

The development of large scale applications requires modular approaches such as Service Oriented Computing [11, 12] or Component Based Software Engineering [9]. In both approaches, the success depends on the availability of: expressive languages for the elements (services and components) and their composition [2, 11], tools for checking the correct usage of the elements, and tools for managing reliable element libraries. As a partial answer, we propose a component model to describe services and service composition. One component offers services which may be called by another component service. We also study the means for verifying the correct usage of these services. Indeed, it is important to detect the defects which could lead to a faulty behaviour of a 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 the desired properties (correctness, compatibility, composability...) we need formal descriptions of the services. In our model, the service providers are explicitly represented as components but service implementation is left out. 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 EJB and CORBA-based approaches [6] are not dealt with.

Our approach is based on a simple formalism for modelling and composing services and components. A component is viewed and used only through its services which constitute its behavioural interface. The use of service is central to

the verification of compatibility when assembling components because the components are "connected" by their services. The contributions of this work are: a formalism that allows a flexibility in the description and the composition of component and services: optional behaviours, optional sub-services and renaming; service interfaces enriched with behavioural informations that contribute to the early detection of service incompatibilities; the verification of behavioural compatibility of service composition.

This article is organised as follows. Section 2 overviews the kmelia formalism. Section 3 describes the composition of services in our formalism and the desired properties. Section 4 details the verifications that are done during the composition. We conclude by a short discussion in Section 5.

## 2 The Kmelia Model

The Kmelia model is a component model based on services. A Kmelia specification describes services, components and compositions. A service encodes a functionality. A component provides services and may require other services (to supply the provided services). A composition links a set of components by their services.

The main features of Kmelia are: the encapsulation of services in components; the enrichment of interfaces that allows an early detection of service composition errors; the flexible description of the service behaviour.

### 2.1 Component, Services and Sub-services

In Kmelia the service is the basic concept (services perform functions) while the component is a modular structuring unit that encapsulates services and allows a fine control of when and by which services they can be called. The separation between services and components allows system models with partially supported services (some services work and others do not). The component defines a shared entity for services (namespace, data, sub-services, constraints). The component provides an *interface* made of *provided services* and *required services* (from some abstract service provider). Consequently the component defines internal services.

Kmelia defines some minor services (called *sub-services*) that can only (and optionally) be called during a transaction with another service. These sub-services can be shared in a component. Sub-services can be used in place of parameters or to introduce some flexibility in a service protocol. For instance in Figure 1, the `main` service requires a `chat` service that can ask for a password. The use of a `passw` sub-service has several benefits:

- it separates the password protocol from the main service protocol, enhancing readability and reuse;
- it allows the control of the points where the password can be asked;
- it can be easily replaced by another password protocol;
- it allows the service to work even if the password is not asked.

### 2.2 Enriching Interfaces of Services

The interface of a service `serv` is made of a signature, a precondition, a postcondition and sets of service names (*subprovides*, *extrequires*, *calrequires*...). The *subprovides* set contains the names of the sub-services.The two others are two kinds of required services names. In Kmelia we distinguish two kinds of service dependencies: external and caller dependencies. The former (*extrequires*) is quite usual and establishes that a service needs other ones in order to work. The former (*calrequires*) is explicitly required from the caller of the service `serv`.

Following is the code of the interfaces of the service `main` which requires a `chat` service and provides a sub-service `passw`, which in turn needs an identification service (`idserver`) from its caller.

```
Service main
Interface
   subprovides : {passw}
   extrequires : {chat}
...
end
```

```
Service passw : String()
Interface
   calrequire : {idserver}
...
end
```

### 2.3 Specifying Flexible Behaviours with eLTS

A service is formally specified with a 5-uple $\langle \sigma, P, Q, V, \mathcal{B} \rangle$ where $\sigma$ is the service signature, $P$ is the precondition, $Q$ is the postcondition, $V$ is the set of local variables and $\mathcal{B}$ is the extended labelled transition system (**eLTS**) which describes the service behaviour.

The service behaviour $\mathcal{B}$ provides details on the interactions between services and the order in which these interactions may occur. Formally $\mathcal{B} = \langle S, \mathcal{L}, \delta \rangle$ where $S$ is the set of states, $\mathcal{L}$ is the set of transition labels and $\delta \in S \times \mathcal{L} \rightarrow S$ is the transition relation. The labels on transitions (which concrete form is: `is--lab-->fs`) are combinations of actions which may be internal actions or interactions. An internal action is a computation (*elementary action* or a composition of internal actions) that does not involve other services. An interaction denotes an exchange on the service link (service channel). In Figure 1, the composition defines such a link between the required service `Client.chat` and the provided service `Server.chat`.

The syntax of an interaction is: `channel(??|!!|!|?)message(param*)`; it is inspired by the Hoare's CSP language. The message can be a service call (??), a service result (!!) or a synchronous communication (send !, receive ?). When writing a behaviour, one does not know which components will communicate, but one has to know the channel on which it will take place. The channel is defined when the components are composed but its name depends on the service interface. The placeholder keyword `CALLER` is a special channel that stands for the channel opened for a service call, otherwise it is the required service name. The following descriptions are the behaviours of the services `main` and `passw` in the Kmelia syntax.

```
Service main
Variables  # local to the service
  c:Boolean
Behavior
init e0    # e0 is the initial state
final e6   # e6 is a final state
{ e0 -- _chat!!chat --> e1,
      # invocation of the chat service on its channel (named chat)
  e1 -- _chat!login(myLogin) --> e2,    # sending the login
  e2 <passw>, # specifies callable sub-services on node e2
  e2 -- c:=_chat?cnx --> e3,
      #ask for the result of the connection
  e3 -- [not c] nop --> e5,
  e3 -- [c] _chat!message("hello world") --> e4,
  e4 --  _chat!message("\stop") --> e5,
  e5 -- _chat??chat --> e6 #wait for end of service chat
}
end

Service passw () : String
# gives the client's password to a trusted server
Variables # local to the service
  trustserv : boolean,       #is the server a trusted one ?
  id : integer
Behavior
init e0    # e0 is the initial state
final f    # f is a final state
{ e0 -- __CALLER!!idserver --> e2,
  e2 -- __CALLER??idserver(id) --> e3,
      # calls the identification service of the caller
  e3 -- trustserv:=isTrusted(id) --> e4,
      # isTrusted is an elementary action here
  e4 -- [trustserv]__CALLER!!passw(myPwd) --> f,
  e4 -- [not trustserv]__CALLER!!passw("") --> f
      # sends the code as a result of the service
}
end
```

The `_chat!!chat` is a (required) service call on the implicit `chat` channel. The `__CALLER!!passw(myPwd)` is a result of service. You can see a notable difference between the above code and the eLTS of Figure 1: all channels are explicit. In many cases, channels can be omitted and deduced either from the context or from default rules. This syntactic sugar is not currently implemented in our prototype.

In an eLTS the states may be annotated with sub-services. It means the sub-services may be launched from this state and the control returns to this state when the launched sub-service is terminated; for example calls to the service `passw` are enabled only in the third state of the `start` service. Such a notation allows flexibility (optional sub-services), service sharing and LTS size reduction.

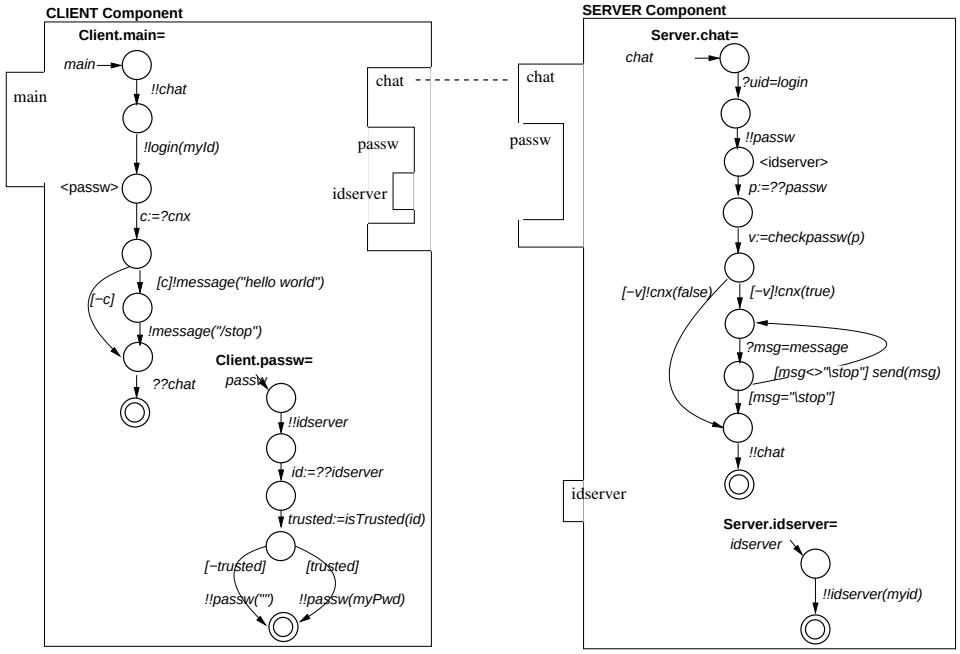


**Fig. 1.** Composition of chat services of two components

## 3 Service Composition

In Kmelia we distinguish two kinds of composition. The first one encapsulates services in a single unit while the second one composes services from different components.

*Encapsulation of Services in a Component* A component that encapsulates several services should have the following properties: at least one service should be provided and it should be visible; every service of the unit should be callable, either directly or indirectly (via another service); every sub-service in a service interface must be callable at some point during the service evolution; no service can have itself as a sub-service; the sets of external required services and caller required services of a service must be disjoint.

*Composition of Components* Two or more components may be composed by linking their provided and required services to build larger components. Therefore a composition of services is the result of the link of the required services and provided services from different components. The composition of services is the support for interaction between the components. It is only at composition time that the channels are resolved according to the links declared between the components. Each linked service is then defined as the communication context of an other.

For the composition to be effective, several properties should be satisfied: *(Static) Interoperability properties*: compatibility of signatures and interfaces (naming, typing and compatibility of pre and post conditions); *Behavioural compatibility*: absence of deadlocks in the communication.

## 4 Behavioural Verification of Services

Formal verification of services may be performed according to various aspects. We generalise the three levels of interoperability of Yellin&Strom [13] to four levels of conformity: service signatures, enhanced service signatures (sub-services may be participant of a service), contracts (pre/post conditions) and behaviours (the correct interactions between the caller service and the called service). In the following, we focus the verification on static interoperability (level 1 and 2) and behavioural compatibility (level 4).

### 4.1 Interface Compatibility

The first step of the verification of the behavioural compatibility is a pairwise comparison of behavioural interfaces. In Kmelia 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 service interfaces. In a similar way, these informations are available for the service descriptions. Accordingly, the static analysis of the interface of a component is achieved by using: *i)* simple correspondence checking algorithms and possibly standard typing algorithms; *ii)* deep investigation on the availability of required or called sub-services. At this stage, we can detect several incompatibilities such as missing sub-services, signature mismatch...

### 4.2 Behavioural Compatibility

The *behavioural compatibility* between services is a widely studied topic [13, 7, 3]. 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 [3].

In Kmelia the behaviour of the component relies on the behaviours of its services. Therefore the component interacts correctly with its environment if its services are *compatible* with the other services. The main concern is to check that a given service interacts correctly with another one (which may be provided by a third party developer). The interaction between services may involve not only two but many services. But we consider only one caller service and one called service at time. Remind that each service is described with a transition system; the transitions are labelled with: service calls, elementary actions, guarded actions and communication actions (messages).

A service is *behaviourally compatible* with another one if their eLTSs are matching: they evolve independently or they perform complementary communication actions. That is the basis of our compatibility analysis approach. This approach is the widely used one [13, 3, 4]; but we adapt it to a more expressive LTS used in our model. A complete interaction between the services of several components results in a pairwise local analysis between the eLTS of a caller and that of the called service. Indeed, two eLTS interact until a terminal state if their labels are in correspondence according to a protocol that we have defined. The protocol is a set of rules based on the labels of the transitions available from a current state. The rules indicate the correct evolutions according to the current states of involved services and the labels of the transition: either independent evolution or a communication involving complementary actions from the peers. After a final state of a called service, the caller may continue with independent transitions or with transitions that imply other (sub-)services.

The ideas presented here are put into practice with the LOTOS/CADP Toolbox [8]. We defined a policy to translate our service behaviours into LOTOS processes. In this translation, all interactions are considered as communications between processes. Thereafter we use the LOTOS communication facilities to deal with behavioural compatibility. A similar experiment has been conducted with the MEC[1] tool.

## 5 Discussion and Perspectives

We have presented an abstract component model and a formalism that permit the flexible description of interacting services which are defined as extended labelled transition systems. This model supports service composition and component composition. A *chat* example involving two components is presented and it illustrates an interaction between composed services and their accompanying sub-services.

Unlike most of existing approaches [13, 3, 5] where a component has a behaviour (called a protocol), we argue for a model were the provided services, defined as LTS, have their own behaviour. This moves up the granularity for the use of components and increases the usability of components by considering a service level. When our service behaviours are reduced to combinations of messages, we get the low level of usability found in the aforementioned approaches.

The study of compatibility at the component behaviour level is central to CBSE approaches and has motivated number of works [13, 7, 3, 5] and applications to web-services [4]. We build on these works but we extend the study to encompass the granularity considered here for services and components. Our approach allows for a local verification of the behavioural compatibility between composed services. Experiments are performed with the approach using existing toolboxes. Compared to related works [3, 10], our approach works at the abstract specification level, it offers a more flexible formalism than the ones proposed by [13, 3, 4] for the description of interacting services. We adopt a pairwise verification approach that avoids state explosion like in [3]. We can extract several

collaborations a la Yellin&Strom [13] from a single of our service behaviours which interweaves collaborations on different channels and allows optional calls of services.

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 to cover mechanised analysis concerns. The prototype already integrates parsers, translators to LOTOS and MEC, static and dynamic interoperability checkers. However we lack a graphical interface to guide and assist the user. Then we will propose an open source delivery of the toolbox.

## References


1. P. Crubillé A. Arnold and D. Bégay. *Construction and Analysis of Transition Systems with MEC*. AMAST Series in Computing: Vol. 3. World Scientific, 1994. ISBN 981-02-1922-9.
2. M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, editors. *Service-Oriented Computing*. ACM, 2004.
3. P. 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.
4. L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *TES*, pages 15–28, 2004.
5. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
6. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to corba objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.
7. 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.
8. 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.
9. 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.
10. 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.
11. M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISE*, pages 3–12. IEEE Computer Society, 2003.
12. M. P. Papazoglou and D. Georgakopoulos. Introduction to service-oriented computing. *Commun. ACM*, 46(10):24–28, 2003.
13. D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.