Electronic Notes in Theoretical Computer Science 189 (2007) 5–20 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Adaptation for Hierarchical Components and Services

# Pascal Andr´e, Gilles Ardourel, Christian Attiogb´e [1](#_bookmark0)

*LINA CNRS FRE 2729*

*University of Nantes France*

Abstract

Software coordination and adaptation are tightly related to modular software entities and access points. These entities (components or services) may be complex, dissimilar (various models) and designed at differ- ent granularity levels. In order to allow interoperability we need rich interface descriptions including service hierarchisation, flexible declarations and precise specifications. In this article we present a Hierarchical Behavioural interface description language that enables the structuring of services, their encapsulation and it also facilitates the use of component interfaces. We also investigate in this work the adaptation for Hi- erarchical Behavioural IDL. We recall various adaptation problems and we introduce modelling techniques and some solutions within hierarchical context considering precision of the interfaces, their layering and flexibility.

*Keywords:* Adaptation, Components, Services, Behavioural Interface Description, Hierarchisation

# Introduction

Component-based development is the process of building programs by gluing to- gether active software parts [[12](#_bookmark28),[19](#_bookmark35)]. Usually the glue adheres on access points and when it does not, one uses adaptation techniques to make it adhere. Software adap- tation [[27](#_bookmark43),[16](#_bookmark32),[8](#_bookmark24)] includes the detection of interaction mismatches and their correction when it is possible. The correction can be either a dynamic adaptation at running time or the insertion of static adaptors (or transformers).

Adaptation may be considered from various perspectives. From the *component* perspective (the Component Based Software Engineering approach) [[26](#_bookmark42),[18](#_bookmark34),[5](#_bookmark21),[15](#_bookmark31)] the access points are interfaces, ports, services or operations.

From the *service* perspective (the Service Oriented approach) [[20](#_bookmark36),[21](#_bookmark37),[6](#_bookmark22)] the access points are interfaces, services or operations.

1 [pascal.andre@univ-nantes.fr](mailto:pascal.andre@univ-nantes.fr) , [gilles.ardourel@univ-nantes.fr](mailto:gilles.ardourel@univ-nantes.fr) ,[christian.attiogbe@univ-nantes.fr](mailto:christian.attiogbe@univ-nantes.fr)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2007.05.045

In a wide acceptance, a software architect would integrate software components from any provider and therefore with a non-restricted range of models. It means that the components can be components *`a la* CBSE or services, assuming that there are many different component models and many service models. In such a context, the software architect needs a language that helps her/him either to define clearly what she/he needs or to find components on the shelf and appropriate adaptation mechanisms. Usually, such a language applies at an interface level and should be

* *abstract, expressive* and *formal* to hide model specific features and implemen- tation considerations; to provide enough information for both the component designer and the component client (the architect); to ensure consistency and to support the verification of properties such as service or component composability;
* *flexible* to allow partial use of components, partial descriptions of services, op- tional use of subservices,
* *scalable* to allow the combination of small elements (services, components, adap- tations) into higher level components.

However, a component model may provide interfaces which are not restricted only to simple synchronous call/response operations; interacting services may be provided through the component interface, this leading to hierarchically structured interfaces. Furthermore, support for adaptation facilities increases the component model reusability.

In this article, we address these needs with a Hierarchical Behavioural Interface Description Language (HBIDL) that is used for our Kmelia component model [[3](#_bookmark17)]. For (re)usability purpose we investigate adaptation issues for such a model and propose some adaptation techniques.

The article is organised as follows: Section [2](#_bookmark1) motivates and presents the hi- erarchical behavioural interface description language of our component model. In Section [3](#_bookmark8) we consider adaptation problems that are either introduced by the hier- archisation level or solved using specific features of our model. We conclude with the perspectives of this work in Section [4](#_bookmark16).

# Hierarchical Behavioural IDL using Kmelia

This section introduces an extension to *Behavioural Interface Deﬁnition Languages* (BIDL) in order to handle complex service interactions and service composition, called a *Hierarchical BIDL*. The section motivates the use of hierarchisation for component documentation, service composition and service adaptation. The Kmelia language is used to illustrate HBIDL capabilities.

* 1. *Structuring Component Interfaces*

Formal and abstract descriptions are valuable to design and reason on CBSE system, especially at the interface level [[24](#_bookmark40)]. We assume any component model with provided and required services in which the *interface* specifies the component interactions

with its environment [[1](#_bookmark18),[18](#_bookmark34)].

An IDL is commonly used for component interoperability. The IDL describes the signatures of provided services. However more details are required (1) on the required services to get modular components, (2) on possible contract definitions,

(3) on the ordering of service invocations. The use of *Behavioural IDL* (BIDL) meets the requirement (3).

In the BIDL approaches [[27](#_bookmark43),[9](#_bookmark25),[10](#_bookmark26),[7](#_bookmark23),[22](#_bookmark38)], the interface specifies the ordering of ser- vice invocations and the *dynamic behaviour* using *protocols*. A protocol specifies the valid interactions between components. For example a *protocol* can be a state transition system [[22](#_bookmark38)], a regular expression [[23](#_bookmark39)], or a non-regular process type [[25](#_bookmark41)]. A protocol may be associated to a component, to an interface or to a connector [2](#_bookmark2) . In the first case the protocol merely controls the component lifecycle (like a pro- cess if there is only one protocol). In the second case the protocol controls the component interactions in some identified relations: an interface can hold on peer- to-peer channels (one per connected component) or on view (like database views). In the last case the protocol controls the communications on a structural access point: it manages the communication aspects of the components or it can be an explicit adaptor. The semantics and usage are slightly different from one approach to another, especially if we have in mind the adaptation issues.

A common characterisation of the existing BIDL approaches is to consider ser- vices as atomic operations and service calls as message sends on an implicit or explicit channel. Such services are defined by a signature (name and parameters) and –if the component model supports assertions– a contract (pre/post conditions). But a service may be more complex than a simple message call: it can handle com- plex functionalities, it can require multiple interactions and it can also call other services.

*Hierarchical BIDL* (HBIDL) is a solution to introduce complex services.

A HBIDL is characterised by the fact that services are first class entities: (1) services may be defined by a dynamic behaviour (a protocol) in addition to their signature and contract and (2) services may be composed of other services. Each service has an enhanced service interface which includes a service dependency com- posed of the provided services and the required services which are used in its context. Therefore, a HBIDL should support component protocols, component composition, service protocols and service composition.

The advantages of using a HBIDL are manyfold. It supports the definition and documentation of complex interacting services. The client-side documentation of a component has detailed information on the service usage. Indeed, the client gradually discovers the documentation of the components and services. First, the component interface shows the services that are accessible at a component level; then the specific interface of each provided service; therefore the client only gets step by step what she/he needs.

Furthermore, assembling components in HBIDL focuses on services, consider-

2 A short comparison of protocols in component models is given at

[lina.atlanstic.net/fr/equipes/team10/Kmelia/](http://lina.atlanstic.net/fr/equipes/team10/Kmelia/)

ing them as functional connecting points rather than structural connecting points (sometimes called gates or ports). This is a convenient view when one tries to fulfil a complex required service.

HBIDL can smoothly support the notion of *compatibility level* (see below) for component connection because it induces a layered presentation of the interface: IDL only, IDL with protocols, IDL with hierarchy and protocols. Therefore com- patibility checking is preserved by HBIDL.

Last, HBIDL due to its emphasis on services, can be a gateway to service oriented models in order to compose components and services (in the sense of the Service- Oriented Approach).

Defining compatibility levels helps when handling heterogeneous models. For example, Becker and al. consider four compatibility levels: syntax, behaviour, syn- chronisation, and quality of service [[4](#_bookmark20)]. The compatibility of services described with an HBIDL can be defined at five levels: signature matching, enhanced service in- terface conformity (including subservices), contract fulfilment, behavioural compat- ibility (the interactions -waiting for data, synchronisation- between the caller and called services are correct) and quality of service (non-functional requirements). For example, if a component A with an IDL interface only (e.g. a component *`a* *la* CORBA Component Model) is composed with a component B with an HBIDL interface (e.g. a Kmelia component) then their compatibility can be checked only at the signature level.

The use of the previous compatibility levels impacts on the adaptation tech- niques. One can either restrict the adaptation at one or more compatibility levels or extend the adaptation to cover the missing compatibility levels in the considered components.

* 1. *Hierarchies of Components and Services in Kmelia*

The Kmelia model [[3](#_bookmark17)] is a simple, formal and abstract component model based on services. It uses the HBIDL features exposed in the previous section. The simplicity relies on the few number of concepts that are used to describe the components and their assemblies. The notion of service is central to Kmelia and a component inter- face describes mainly services. This means that the components are connected via their services (functional connections). The interaction model is therefore simple: required services are directly linked to provided services. A component specifica- tion language named Kmelia and a prototype toolbox (COSTO: Component Study Toolbox) support the Kmelia model and property verification. The remainder of the section is an overview of the Kmelia model illustrated on a bank Automatic Teller Machine (ATM) example and its withdraw service.

A Kmelia component is defined through an abstract state model (made up with variables, an invariant, and an initialisation), an interface (made up with provided and required services) and a constraint definition (logic expressions). Figure [1](#_bookmark3) shows an overview of a specification of an ATM core component in Kmelia.

Basically, a Kmelia service encodes a functionality; it is defined with an interface and a behaviour. The service interface includes the service signature, the local dec-

COMPONENT ATM\_CORE

/\* The ATM\_CORE component is the central component for a bank cashier station.

...

\*/ INTERFACE

provides : {withdrawal, account\_query, deposit, transfer} requires : {ask\_authorization, ask\_account\_balance}

TYPES

CashCard : struct {code:Integer, id:Integer, limit:Integer} // record type CONSTANTS

// constants definitions

...

VARIABLES

// variables definitions name : String, swallowed\_cards : Set, available\_notes : Integer

...

PROPERTIES

// predicates

cash\_disp: available\_notes >= 0

...

INITIALIZATION

// variables assignments

...

SERVICES

provided withdrawal (card : CashCard)

// see the service withdrawal in Figure 2

required ask\_authorization (id : Integer, code : Integer) : Boolean

...

end

Fig. 1. Overview of the Kmelia component syntax

larations, the assertions (pre/post conditions) and the service dependency (related to service composition). The service dependency includes the references

* + 1. to provided subservices: they are the services which are provided in the context of another service
    2. to required services. The latter are required from the component itself, from the calling component or from any components. Figure [2](#_bookmark4) shows a specification of the withdrawal service of the core component for the ATM system in Kmelia.

Provided withdrawal (card : CashCard)

/\* The service withdrawal is available if there is enough money in the cash dispenser.

This services requires a bank credit card, a code, an amount to withdraw. An authorization is required from the bank consortium.

This service provides an identification subservice if needed.

\*/ Interface

subprovides : {ident}

calrequires : {ask\_code, ask\_amount} //required from the caller extrequires : {ask\_authorization}

Pre

available\_notes >= available\_cash Variables

nbt : Integer, // nbt : number of authorized trials of code entering c : Integer, // c : input code given by the user

...

Behaviour init i final f

... // see the service behaviour in Figure 4 Post

available\_notes <= pre(available\_notes)

// (success && (available\_notes = pre(available\_notes) - a)) ||

// ((not success) && available\_notes = pre(available\_notes)) end

Fig. 2. Overview of the Kmelia service syntax

The subprovides, calrequires, extrequires clauses in the interface of the withdrawal service make explicit the hierarchy and the dependencies between services: the withdrawal service offers an ident subservice and requires three other services, two of them being required from the component which is calling withdrawal.

Component assemblies establish the communication channel used by the commu- nication actions. Assembling Kmelia components consists in linking their pairwise services: required services may be linked to provided services. An implicit chan- nel is associated to this link that supports the *communication actions* or messages between the services. The semantics of the links is not straightforward because it must conform to the service interface hierarchy. Indeed the services that appear in the subprovides and the calrequires clauses of the service interface dependency must (i) share a common link (they are sublinks) and (2) their links must conform to the hierarchy levels. This constraint is recursive on service inclusion. A com- ponent composition is the encapsulation of an assembly within a component with a projection of services by promotion links. Promotion links relate the composite services to the inner component services.



as : ATM\_SYSTEM

ac : ATM\_CORE

ui : USER\_INTERFACE

aac : AAC

authorization

ask\_

authorization

withdrawal

ask\_for\_money

account\_update

ask\_ autho

rization

ask\_amount

amount

behaviour

behaviour

ask\_code

ident

code

ask\_id

lb : LOCAL\_BANK

debit eject\_card swallow\_card

display

account\_query

query\_account

balance

deposit

deposit

ask\_ account\_

balance

ask\_

account\_balance

transfer

 provided service required service

service link subservice link promotion link

extrequires

subprovides

service call provided subservice

 service call on

calrequiresthe caller componnent

Fig. 3. Assembly for an ATM System

Figure [3](#_bookmark5) is a graphical view of a Kmelia model for the bank ATM. The as com- ponent is a composition of an ATM CORE (ac) with an ATM user interface (ui). The main provided service behaviour of the ui component drives the user com- mands. For example, the user can ask for money (required service askformoney) which is linked to the service withdrawal provided by the ATM core ac compo- nent. According to Figure [3](#_bookmark5), the withdrawal service may call internal services (debit,...), external services (askauthorization), external services required from the caller (askcode, askamount) and it provides the *ident* service in the context of *askcode*. Note that the amount and code links are sublinks: they share the askformoney-withdrawal link and its implicit communication channel.

The component and service usages are flexible:

* An assembly may be valid for one service only, provided that its dependency chain is fulfilled. For example, all the required services that are only needed by the unused transfer provided service have not to be fulfilled for the ATMSYSTEM assembly to be correct.
* A service can provide optional subservices. For example, the askcode service can be linked to a service that does not need identification (ident service). Similarly the askformoney service can be linked to a withdraw service that does not need code (code service).
* Provided subservices may be included as services with explicit service call or only as behaviours without a service call (see next section).
  1. *Hierarchy of Behaviours in Kmelia*

The hierarchy of service interfaces is reflected in the service behaviour: this permits a precise description of the use of a subservice in the context of the interaction with a service.

In Kmelia, a service behaviour is an extended labelled transition system (eLTS) [[3](#_bookmark17)] where the states define the service evolution steps and the transitions are labelled with possibly guarded combination of actions: [guard] action\*. The actions are either *elementary actions* or *communication actions*. An elementary action, an assignment for example, does not involve other services; it does not use a commu- nication channel.

The communication actions use either the standard communication primitives

! and ? for sending/receiving simple messages or their extended forms !! and

?? to deal with service calls and service responses. The communications are syn- chronous in the context of services running concurrently. They are prefixed with a communication channel which can either denote the required service or the caller. A communication channel to be used in a behaviour has to be established by a link like the askformoney-withdrawal link in Figure [3](#_bookmark5).

Figure [4](#_bookmark7) is a visual representation of the withdrawal service eLTS which specifies its behaviour; this figure is produced by the COSTO toolbox.

In Kmelia, service behaviours may contain execution points (states or transi- tions) where a subservice (declared in the service’s interface) can be called. These states or transitions are annotated with Kmelia’s *vertical structuring operators*. For instance, the label of the node e1 in Figure [4](#_bookmark7) expresses that an optional service ident may be called by widthdrawal’s caller when the running reaches node e1. This label features the <<>> operator that denotes an *optional service call*. Kmelia main vertical operators are:

* *optional service call* : <<>> ,
* *optional behaviour insertion*: <||>,
* *mandatory service call* : [[]],
* *mandatory behaviour insertion*: [||].

These structuring mechanisms provide a means to reduce the LTS size, to share

e2



rep:=\_ask\_authorization!! ask\_authorization(card.id, c)

e3

display("Enteryourcardcode,please") e0

i

; nbt:=3; success:=false

display("Enterthecashamount,

please?")

CALLER!rdv()

CALLER!!ask\_code()

; display("Card swallowed, sorry"); swallow\_card()

; display("Transaction

e5 display("requiretoomuchmoney,

pleaseentertheamountagain")

e6

e1<< CALLER.ident>>

CALLER!!ask\_amount() ; CALLER??ask\_code(c); nbt:=nbt-1

e2i

refused"); eject\_card()

e7

CALLER??ask\_amount(a)

; debit(c, m); eject\_card()

e8

success:=true

e4

CALLER!!withdrawal(success)

f

Fig. 4. The withdrawal service eLTS

common services or subservices and to master the complexity of service specification. All these structuring operators are defined in such a way that formally the unfolding of an eLTS results in a LTS (in a recursive way). The formal semantics

of the structuring mechanisms can be found in [[2](#_bookmark19)].

# Adaptation Problems and Solutions

The following categories of adaptation problems are especially relevant in HBIDL:

* Classic adaptation problems that can be seen through the lens of the new language and benefit from its expressiveness.
* Granularity mismatch in HBIDL: adapting services which hierarchy is organised differently. Having more expression power allows to describe precise constraints relevant for a specific context but that must be adapted when a service or a component is used in a different environment.
* Granularity mismatch between a HBIDL and a BIDL: having a more expressive language does not mean every model uses the expressiveness to the full extent. For a new language to be effective, differences of levels of detail must be dealt with, as well as interoperability with models expressed in languages which intersect with the new one. Problems from this category can be treated like granularity mismatches in HBIDL by considering BIDL models as flat HBIDL models.

In the following we describe the process that is used to detect adaptation prob- lems in a HBIDL architecture and we provide solutions in the Kmelia model.

* 1. *Adaptation Process with Kmelia and COSTO*

The COSTO prototype features several verification algorithms that check compati- bility of components and services at different levels: signature, interface, dependen- cies, behavioural compatibility. Behavioural compatibility is checked by exporting our Kmelia models to Mec [[13](#_bookmark29)] or Lotos [[17](#_bookmark33)] and reusing their respective model checking tools. We assume that the matching between names (of different com- ponents, services or messages) has already been established, either manually or automatically (e.g. by ontology-based approaches).

For each mentioned adaptation problem we use the following pattern.

* + 1. First we indicate the earliest compatibility level at which the problem occurs.
    2. Next we specify how we identify which kind of adaptation problem it belongs to.
    3. Then we explain how to generate systematically an adaptor and verify that it ensures compatibility. Depending on the constraints of the running environment, the adaptor can take several forms: a component inserted between two mismatch- ing components, a proxy service delegating to one of the services, or an alternative interpretation of a behaviour. We will focus here on the first one.
    4. Last, we precise if the adaptation problem could have been avoided at minimal cost using the Kmelia structuring operators.

We believe that designers of components and services should anticipate different uses, provided that their specification language helps them to do so at a minimal cost in design time and verification time. We name *implicit adaptation* this in- herent ability of a service or a component to be compatible with many others by construction.

At a message or service signature level, implicit adaptation focuses on optional arguments, default values, compatible subtypes for arguments and result. At a hi- erarchical service interface level, it includes optional subservices, implicit linking to services or subservices. At a service assertion level, it means that the compatibility between provided and required pre/post conditions is ensured via propagation from the previous levels. At a service behaviour level, all the above adaptations apply together with possible alternative behaviours (w.r.t. observational equivalence: the behaviour of the service is not changed from the point of view of client services) using the behaviour insertion operators introduced in Section [2.3](#_bookmark6).

* 1. *Adaptation in HBIDL: Parameters vs Messages*

This problem is an example of a standard BIDL adaptation problem taken in a HBIDL context. In the *Parameters vs messages* problem, there are different treat- ments for data exchanges between related components: according to a service, one component may use parameters where the other uses messages to send the data value for the service. It is a variant of the *Multiple action correspondence* [[7](#_bookmark23)]. Based on dif- ferent interpretations of an imprecise textual specification such as ”The client must communicate an account number and an amount to the service *deposit* ”, one service could use parameters while another could use message sends. This is illustrated in

Figure [5](#_bookmark9). Given that the correct data is ”communicated”, the communication has to be adapted when a client of the service *deposit* uses a different interpretation.

|  |  |  |  |
| --- | --- | --- | --- |
| ATM\_CORE.deposit(int account,  double amount) =  i  deposit amount on account  deposit |  | | USER\_INTERFACE.behaviour() =  e10  deposit!!deposit()  e11  deposit!account(12)  e12  deposit!amount(100)  e13 deposit |
|  |  |
|  |
|  | |

Fig. 5. Differences in communication of parameters

In Figure [5](#_bookmark9) the behaviour service from USERINTERFACE calls the deposit ser- vice from ATMCORE to credit the account number 12 by 100. The signature of deposit is deposit(int account, double amount) but the call from behaviour does not contain parameters and it is followed by two messages containing the pa- rameters.

The *parameter vs messages* problem is detected at level 1: signature mismatch between the deposit provided service of ATMCORE and the deposit required service of USERINTERFACE.



CALLER?account(ac)

a2

CALLER?amount(m)

a3

ARdeposit!!deposit(ac,m)

a4

ARdeposit **ARdeposit <|>APdeposit**

APdeposit

i

ADAPTER.deposit() =

deposit

deposit amount

on account

i

ATM\_CORE.deposit(int account,

double amount) =

USER\_INTERFACE.behaviour() =

e10

deposit!!deposit()

e11

deposit!account(12)

e12

deposit!amount(100)

e13 deposit

Fig. 6. Adapting differences in communication of parameters

The adaptability is checked in COSTO using a simple graph traversal algorithm which looks for an unavoidable sequence of message sends in the caller’s behaviour that matches the parameters and that takes place between the service call and any other communication with the service. If such sequence is found, the prob- lem is diagnosed as a *Parameters vs messages* and can be solved. Figure [6](#_bookmark10) shows an adaptor component between the deposit provided service of ATMCORE and the deposit required service of USERINTERFACE. This adaptor illustrates another Kmelia operator: channel<|>channel . This operator redirects communications from a channel to another one, thus simplifying the writing of some adaptors that require pre-processing before reverting to normal communication, like an incor- rect order of messages. Furthermore, unidirectional versions of this operator exist

(channel<|channel and channel|>channel). An adaptor can be automatically generated by matching with a sequence of receptions the sequence of calls found in the diagnosis phase, adding a call to the service with its parameters, then adding the redirection operator.

The *parameter vs messages* problem can be further complicated by the use of parameters with types of different granularity in the client and in the provider, for instance the deposit service could have had a structured type in its signature while the caller would have used primitive types in its call. The complexity of the detection and the generation is related to the complexity of the data structures used in the parameters (data might have to be constructed or decomposed) and the similarity metrics used for identifying the parameters. The diagnosis algorithm of COSTO for the *parameter vs messages* problem is parametrised with an identification method which has to be changed to incorporate the variants of the problem.

* 1. *Adapting Hierarchical Mismatch*

When a component expects to call a service which is not hierarchical and is con- nected to one where part of the functionality is described as a subservice, a *Hierar- chical Mismatch* occurs.



ac:ATM\_CORE

withdrawal() =

**withdrawal**

ui:USER\_INTERFACE behaviour() =

i

**<ident>**

e10

**CALLER?amount(m) withdrawal!!withdrawal() withdrawal** e11

withdrawal?identifier(i)

e12

ident() = **withdrawal!amount(10)**

e13

i

**CALLER!identifier(id)**

**ident**

Fig. 7. Differences in communication granularity

Being designed with different granularity levels in mind, the services withdrawal and behaviour from Figure [7](#_bookmark11) are not compatible: in behaviour, identification is a simple communication made in the context of the withdrawal service but the identification is considered to be managed by the ident service in withdrawal’s component, probably because it is meant to be used in the context of several other services of the component.

This mismatch can be detected by a behavioural compatibility analysis using COSTO tools. For instance, the results of the analysis with MEC gives the following information :

transition\_system Verif\_ATM\_CORE\_withdrawal

< width = 2; list = (ATM\_CORE\_withdrawal, USER\_INTERFACE\_behaviour )>

...

deadlock = { e(i.e11) }

unavoidable = { e(init.init), e(init.e10), e(i.e11) }

Fig. 8. Extract of a result from the compatibility analysis using MEC

A deadlock is detected at the state i of withdrawal and at the state e11 of behaviour. This incompatibility can be diagnosed as a *Hierarchical Mismatch* problem because one of the states (i of withdrawal) where the deadlock occurs offers a subservice which starts with the identification message that matches the one following e11 of behaviour. This diagnostic is confirmed by taking the following steps [3](#_bookmark13) :

* A call to the ident subservice just before the identification message is added to a copy of the LTS of behaviour
* The compatibility check is run again. It should give another deadlock cor- responding to the lack of waiting for the result of ident in behaviour (withdrawal??ident).
* The wait for the result of ident is added in the copy of the LTS before the newly-found deadlock.
* If this compatibility check passes, the problem is correctly diagnosed and can be solved using an adaptor.

withdrawal



behaviour() =

withdrawal!!withdrawal()

e11

withdrawal?identifier(i)

e12

withdrawal!amount(10)

e13

withdrawal

ADAPTER.withdrawal() =

a1

arwithdrawal!!withdrawal()

a2

**arwithdrawal!!ident()**

a3

**arwithdrawal?identifier(i)**

a4

**CALLER!identifier(i)**

**arwithdrawal??ident()**

a5

CALLER?amount(m)

ident

arwithdrawal!amount(m)

arwithdrawal

ident

CALLER!identifier(id)

i

ident() =

i <ident>

CALLER?amount(m)

withd

ac:ATM\_CORE

withdrawal() =

ui:USER\_INTERFACE

e10

rawal

Fig. 9. Adapting differences in communication granularity

Figure [9](#_bookmark12) shows an adaptor that solves the problem. The service behaviour from ui is now linked with the withdrawal service of the adaptor, which depends on a required service arwithdrawal that is linked to the withdrawal service of ac. Calls should be read from right to left, following the required-provided links: when it is called, the withdrawal service of the adaptor calls withdrawal from ac; then it calls the subservice ident in withdrawal’s context (hence the channel

3 We use several optimisations reducing the LTS size that are outside the scope of this paper.

arwithdrawal); then it waits for the identification message; then it transmits the identification message to its initial caller (ui’s behaviour).

The adaptor is created by generating a LTS that starts the called service (withdrawal from ac), relays all communication from the caller that happens be- fore the deadlock (in our example we start directly with the deadlock), starts the subservice and relays communication again.

* + First, the LTS obtained from the diagnosis phase is reused (it has been restricted to communications on the withdrawal channel and reduced).
  + The withdrawal channel is renamed with the name of the required service for the adaptor (in our case arwithdrawal).
  + Transitions that call and wait for the subservice are inserted at the points dis- covered in the diagnosis (initial and second deadlocks).
  + Every other communication on the channel is forwarded:
    - A transition is added after every message that is sent by the ini- tially called service (withdrawal) for sending the information to the adaptor’s caller (for instance, CALLER!identifier(i) is inserted after arwithdrawal?identifier(i)).
    - A transition is added before every message that is sent to the initially called service (withdrawal) for receiving the information from the adaptor’s caller (for instance, CALLER?amount is inserted before arwithdrawal!amount).

The behavioural compatibility between ui’s behaviour, the adaptor’s withdrawal and the withdrawal and ident services of ac can be checked using COSTO.

This *Hierarchical Mismatch* can be avoided using implicit adaptation when the creation of the service ident is coming from a refactoring of the service withdrawal. In such a situation the designer should be conscious that while identifying, naming and factorising some part of behaviour are good from the readability point of view, that breaks compatibility with client services that predate the change. The <||> operator is a vertical composition operator that could have been used instead of the <<>> operator in the example. This operator allows to branch to the subservice either by calling it or without the call in order to be compatible with both new clients and older clients [4](#_bookmark14) .

Figure [10](#_bookmark15) illustrates its use with another Kmelia operator: [||]. This *mandatory service or behaviour call* operator is used for describing protocols that control the correct ordering of service calls. We added the login and logout services to the ATMCORE component in order to describe a very simple protocol. The users of the component call the protocol withdrawprotocol then they have to call the other services in its context (i.e under its control) or have interactions that match those of the services under the control of the component.

The <||> and [||] operators are useful for *implicit adaptation*, but they cannot

4 It is not as powerful as using adaptors because it does not work for several classes of services (for instance, services with parameters).

ac:ATM\_CORE\_BIS

withdrawProtocol()

i

[|login|]

withdrawProtocol

withdrawal

ident

e0 <|account\_queryl|> [|withdrawal|]

e1 <|withdrawal|> [|logout|]

f

login logout

account\_query

Fig. 10. Implicit Adaptation in a Protocol

always replace their less flexible counterparts: for instance, if the *mandatory ser- vice call operator* [[]] had been used in the protocol of Figure [10](#_bookmark15), then the services withdrawal and accountquery could not have begun with the same communica- tion action without being ambiguous at the state e0.

# Conclusion

In this article we have presented the Hierarchical Behavioural Interfaces Description Language (HBIDL) that is used in the Kmelia abstract Component Model. The interface hierarchisation appeared through the interfaces of components which are complemented with the interfaces of services.

The Kmelia component model is based on services and it provides several struc- turing mechanisms: horizontal structuring mechanisms especially based on the link- ing of services and their related subservices; vertical structuring operators to enable the description of large services with encapsulated or shared (sub)services. An ex- ample of an ATM was used to illustrate the structuring of component and service specifications with hierarchical interfaces.

We have considered adaptation problems encountered in BIDL and also related to our HBIDL: hierarchy mismatch and parameter vs message mismatch. We moti- vated and showed what solutions are used for these problems in our Kmelia model. The vertical structuring mechanisms help to tackle the adaptation problems. Com- pared to BIDL related works, we have emphasised the impact of the hierarchical BIDL on the adaptation problems and we have shown that the use of structuring mechanisms may simplify design and verification of both services and adaptors.

Overall, we are concerned with the verification of components with respect to their dynamic and functional properties and their preservation after adaptation.

We plan the integration of adaptation techniques into the COSTO prototype which already handles verification of components, services and compatibility of the assemblies. The investigated process is the following: after a failed compatibility verification, a module will attempt to diagnose the incompatibility as an adaptation problem and to generate an adaptor and launch the verification again. This inte- gration of adaptation with verification techniques is an open field: the detection of

mismatches must be connected with the proposition of adaptations and the decision process. A research direction could be the specification of adaptation problems and their solutions in a language that uses HBIDL and coordination patterns.

# References

1. Allen, R. and D. Garlan, *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology 6 (1997), pp. 213–249.
2. Andr´e, P., G. Ardourel and C. Attiogb´e, *Defining Component Protocols with Service Composition*, in:

*6th International Symposium on Software Composition*, LNCS (2007), p. to appear.

1. Attiogb´e, C., P. Andr´e and G. Ardourel, *Checking Component Composability*, in: *5th International Symposium on Software Composition*, LNCS 4089 (2006), pp. 18–33.
2. Becker, S., S. Overhage and R. Reussner, *Classifying Software Component Interoperability Errors to Support Component Adaption*, in: I. Crnkovic, J. A. Stafford, H. W. Schmidt and K. C. Wallnau, editors, *CBSE*, LNCS 3054 (2004), pp. 68–83.
3. Bergner, K., 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*, Cambridge University Press, New York, NY, 2000 pp. 189–210.

URL <citeseer.ist.psu.edu/article/bergner99formal.html>

1. Beyer, D., A. Chakrabarti and T. Henzinger, *Web Service Interfaces*, in: *14th international conference* *on World Wide Web, WWW’05* (2005), pp. 148–159.
2. Bracciali, A., A. Brogi and C. Canal, *A Formal Approach to Component Adaptation.*, Journal of Systems and Software 74 (2005), pp. 45–54.
3. Brogi, A., C. Canal and E. Pimentel, *On the Specification of Software Adaptation* (2003), in FOCLASA’03, ENTCS, 90 (in press).

URL <citeseer.ist.psu.edu/brogi03specification.html>

1. Canal, C., *On the Dynamic Adaptation of Component Behavior*, in: Canal et al. [[11](#_bookmark27)], pp. 81–88, iSBN

: 84-688-6782-9.

1. Canal, C., L. Fuentes, E. Pimentel, J. M. Troya and A. Vallecillo, *Adding Roles to CORBA Objects*, IEEE Trans. Softw. Eng. 29 (2003), pp. 242–260.
2. Canal, C., J. M. Murillo and P. Poizat, editors, “Issues on Coordination and Adaptation Techniques: Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT’04),” Technical Report, Oslo, Norway, 2004, iSBN : 84-688-6782-9.
3. Gelernter, D. and N. Carriero, *Coordination Languages and their Significance*, Commun. ACM 35

(1992), p. 96.

1. Griffault, A., “Contribution `a l’´etude des syst`emes communicants et des algorithmes d’exclusion mutuelle,” Ph.D. thesis, Universit´e de Bordeaux I (1989).
2. Gschwind, T., U. Aßmann and O. Nierstrasz, editors, “Software Composition, 4th Int. Workshop, SC 2005,” LNCS 3628, Springer, 2005.
3. Heineman, G. T., 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,” LNCS 3489, Springer, 2005.
4. Heineman, G. T. and H. Ohlenbusch, *An Evaluation of Component Adaptation Techniques* (1999), technical Report WPI-CS-TR-98-20, Worcester Polytechnic Institute, February.

URL <citeseer.ist.psu.edu/article/heineman99evaluation.html>

1. LOTOS, I., “A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour,” International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1988.
2. Medvidovic, N. and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering 26 (2000), pp. 70–93.
3. Papadopoulos, G. A. and F. Arbab, *Coordination Models and Languages*, in: *761*, Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 1998 p. 55.

URL <citeseer.ist.psu.edu/article/papadopoulos98coordination.html>

1. Papazoglou, M. P., *Service-Oriented Computing: Concepts, Characteristics and Directions*, in: *WISE*

(2003), pp. 3–12.

URL [http://ieeexplore.ieee.org/xpls/abs all.jsp?arnumber=1254461](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1254461)

1. Papazoglou, M. P. and D. Georgakopoulos, *Introduction to Service-Oriented Computing*, Commun. ACM 46 (2003), pp. 24–28.
2. Pavel, S., J. Noy´e, P. Poizat and J. Royer, *A Java Implementation of a Component Model with Explicit Symbolic Protocols*, in: Gschwind et al. [[14](#_bookmark30)], pp. 115–124.
3. Plasil, F. and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on SW Engineering 28 (2002).

URL <citeseer.ist.psu.edu/plasil02behavior.html>

1. Poizat, P., J.-C. Royer and G. Salau¨n, *Formal Methods for Component Description, Coordination and Adaptation*, in: Canal et al. [[11](#_bookmark27)], pp. 89–100, iSBN : 84-688-6782-9.
2. Su¨dholt, M., *A Model of Components with Non-regular Protocols*, in: Gschwind et al. [[14](#_bookmark30)], pp. 99–113.
3. Szyperski, C., “Component Software: Beyond Object-Oriented Programming,” Addison Wesley Publishing Company, 1997.
4. Yellin, D. and R. Strom, *Protocol Specifications and Component Adaptors*, ACM Transactions on Programming Languages and Systems 19 (1997), pp. 292–333.