JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 4
Special issue: TOOLS USA 2003

Design by contracts: Analysis of hidden dependencies in component based applications

D. Enselme, G. Florin, F. Legond-Aubry, Conservatoire National des Arts et Métiers, Paris, France

Components are increasingly used to create complex and distributed systems and applications. They are often viewed as simple servers, which limits their capacity for collective action. In this paper, we propose a method to simplify their assembly and their potential re-usability. We use the notion of dependency and contract between components to explicitly design an entity that guarantee the correctness of the built system. We introduce split contracts and delegations of properties to check, both at conception and execution time, the correctness of the built system. Our solution increases the independence of the participating entities by isolating the core components and transferring the aggregation into specific "glue" components.

1 INTRODUCTION

The Object Oriented approach was introduced to offer powerful tools and efficient structural design. Classes offer a clear hierarchical organization. However complexity remains in the interactions that tightly bind each object with the others. This hinders the development of large scale industrial applications. Components were introduced to enhance the isolation and the separation of concerns by increasing the granularity of the manipulated entities, and by giving them new capabilities. But if encapsulation gives abstraction power, it hides the specification of the component (and mainly its internal behavior). Components are black boxes without a "user manual".

Contracts were introduced in Objects by Helm [5]. This aimed to compensate the lack of methods to express relations between objects. They were used to specify behavioural compositions. Using contracts provides an orthogonal dimension to the one provided by the class structure. Techniques of development are more and more based on the component approach [3]. To improve these techniques and allow reuse, contracts were extended and adapted to them by Meyer [10] and Jezequel [6]. Usually, contracts are only associated to servers. That limitation is still present in contracts models. This is the significant anti-symmetry of the call because only servers imposes its conditions of use without even considering the clients nature.

This work[1] offers developpers an integrated development environment with anal-

[1]This work is sponsored by the French government research contract RNTL ACCORD. http://www.infres.enst.fr/projets/accord/

Cite this article as follows: D. Enselme, G. Florin, F. Legond-Aubry: "Design by contracts: Analysis of hidden dependencies in component based applications", in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pages 23–45, http://www.jot.fm/issues/issue_2004_04/article2

ysis tools that use more symmetric contracts to describe composition of components. This paper begins by stating current trends in the use of the contracts for components. Then we introduce a new point of view for the call of a component service. From this point, we set up contracts for this type of call and define a notion of compatibility between two contracts. From this basic definition, we then extend the compatibility notion to n components and define all the possible dependencies. Finally, we apply these notions to verify the integrity of an application developed with these contracts. To illustrate our approach, we present a simple cash dispenser example.

2 TOWARDS A COMPATIBILITY RELATIONSHIP

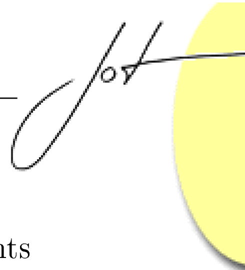
Components and contracts

A component is a cooperative composite entity similar to those described in Architecture Description Languages [8]. A component has multiple interfaces which are sets of operations - also called methods. Sometimes interfaces are grouped to make ports which become a point of interaction. A first attempt to introduce contracts in components was done in 1999 in the article “Making Component Contract Aware” [2]. It introduces four types of contract : syntactic, behavioral, synchronization, and quality of service. It specifies conditions and a unique contract carrying both client and server constraints.

An entity that interacts can only accept the contract (or one of its siblings) and respect it. This point of view limits the assembling capacity of a developer because some entities could have different compatible specifications without having any obvious relationship - i.e. same contract. Another restriction is that the contracts have to be used during execution in order to check the interaction validity. Finally, they are dedicated to an application and a specific platform even if adaptation remains possible. Contracts are made for a specific context therefore it is difficult to extract a component from the whole.

Tools like Jcontract [14], Icontract [18], Eiffel [9] implement “Design By Contract” assertions but for an Object Oriented Model not for components. Moreover they are used for test purposes and not at all for model checking.

To relieve these “dependence” and “checking” limitations, we choose to use “split” contracts. Each side has its own requirements and its own guarantees expressed as a set of properties. The client defines a set of required properties and the server a set of offered properties. If two entities have to interact, both split parties set a contract for the interaction. We are currently independent from a specific platform model like COM+, EJB [1], CCM [13], .NET [11]. All notions will be kept as abstract as possible. Though, In our model, stress is put on the semantic and the pragmatic viewpoints that is to say respectively on the functional and non-functional properties of a component. This document mainly deals with the description of interactions



between components using contracts but wholly viewed as collaboration [15].

Contracts define dependencies in relations and interactions involving elements of the application. We associate, as in the CORBA CCM Model [16], to each interaction, specifications which accurately set the required service context (the client point of view) and the offered service context (the server point of view). Specifications are expressed by pre and post-conditions. Assertions determine the guarantees and the obligations applied to each participant in interaction. Server and client have to specify required conditions that the server and client must provide. If the server or client do not provide them, the binding won't be established.

Compatibility and sub-typing

Firstly, we introduce a compatibility notion between two components which leads us to define a compatibility relation between services (resp. interfaces, operations). Roughly speaking, this notion enables the substitution of one component (resp. service, interface, operation) by another one. A classical compatibility notion is given by the sub-typing notion : a type T is a subtype of a type T' (noted $T <: T$ ') if each value of T can be used in a consistent manner for each expected values of T'.

Our compatibility notion is based on an extension of sub-typing. To define the compatibility between two operations, we extend the classical sub-typing notion. We denote $PO <: RO$ the compatibility of a Required Operation (RO) and a Provided Operation (PO). To check this newly defined compatibility, we add a specification to the involved operations. These specifications are composed of a set of properties. From now on, the properties describe the operations. We set S_{PO} (resp. S_{RO}) as the specifications of PO (resp. RO).

Services are too often under-specified but this is the only data (aside testing) that we can rely on and programmers always use the specifications for the important parts of the new designed component. So when we talk about compatibility between PO and RO, we mean compatibility between their respective specifications: $S_{PO} <: S_{RO}$.

Moreover, we can distinguish different levels, or so called point of view, in the specifications. Compatibility can be considered from many viewpoints. We use a classification deduced from the Natural Language Processing point of view. There is the syntactic level which deals with the signature and the manipulated data. The semantic level which deals with the functional properties of the service. And the pragmatic level that deals with the difficulties raised by the component environment and the way it is used. Usually the compatibility relationship uses sub-typing. We enforce that an operation o of type T is semantically compatible with an o' operation of T', if they have the same number of parameters and the same identifier, if the contra-variance of *in* parameters and the covariance of *out* parameters are confirmed. The contra-variance of *in* parameters asserts that they are in reverse order from the sub-typing relation. The covariance of the *out* parameters asserts that the

parameters carry out the same sub-typing relation order.

The following two definitions illustrate the contra-variance of the *in* parameters for a provided and a required operation:

- `Provided Operation :  void an_operation (in long parameter)`
- `Required Operation :  void an_operation (in int parameter)`

The following two definitions illustrate the covariance of the *out* parameters for a provided and a required operation:

- `Provided Operation :  void an_operation (out int return_param)`
- `Required Operation :  void an_operation (out long return_param)`

This prohibits the sub-typing relationship for the *inout* parameters. This definition is one possibility among many others but it has the advantage of limiting the semantic variations of operations having the same signature (syntactically compatible) but with a totally different semantic. There is no automatic solution. Only the human brain can make the difference. As this compatibility relationship relies on sub-typing notions, the relation is transitive but not symmetric. So we have $\neg(PO <: RO \Rightarrow RO <: PO)$.

Then comes the semantic compatibility. In component interaction, the specification S_{RO} of a Required Operation (RO) is a set of properties required to prove the correctness of the client. In a similar manner, a provided operation (PO) and its specification S_{PO} guarantees the "usage" properties of all correct implementations of a server. Figure 1 illustrates this issue.

A call correctness can be asserted by determining the "compatibility" conditions in which a client wishing to use a service RO could use a PO service instead. So, the call correctness is stated as conditions to satisfy for RO to be replaced by PO. We choose to use the pre and post conditions formalism. In a non-distributed monolithic programming context, assertions are expressed in classic first order logic. We don't need a specific expression of knowledge because the process has a global vision of itself and its state. However in distributed, concurrent programming, there is no global state, there are only some partial views of the systems. Making a non-outdated global view is difficult and costly in many ways. It is therefore imperative to introduce a knowledge expression language (an epistemic language) to reify the state of knowledge of the entities. Another point is the fact that elements are concurrently addressed so a temporal expression language is also necessary. Therefore, in the concurrent distributed environment of the components, we set a pre-condition (O_pre) as a modal logic predicate that can be temporal and/or epistemic [17]. This

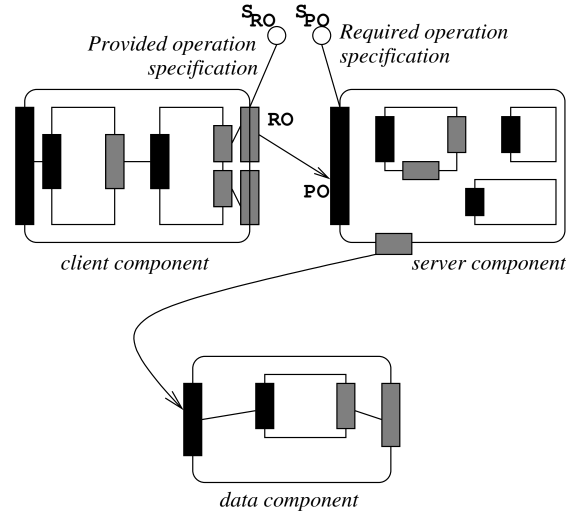


Figure 1: Interaction between a provided and a required service

predicate is evaluated just before the execution of the operation and sets the conditions of the proper realization of the interaction (and subsequently the service). In most cases, the constraints do not specify one unique coherent state but a family of acceptable execution historic that will lead to an acceptable execution. In the same way, a post condition (O_post) defines a set of potential correct futures (after the correct achievement of the service).

A compatibility definition

The specification of an operation compels to use a temporal logic. We choose to use the Temporal Logic of Action [7]. In TLA, an execution is viewed as a sequence of steps, each producing a new state by changing the values of one or more variables. We apply the same analogy for components and we consider an execution to be the resulting sequence of a succession of states that will take the semantic meaning of the studied component interactions. At the very instant where the operation is enabled, the O predicate is true. The same holds for the $O \wedge O_pre$. After the O execution, the operation following O (noted $nextO$ or XO) is enabled. The predicate $XO \wedge O_post$ is true at this very moment and the post conditions are verified. This invocation specification presumes the atomicity of the operation from the client point of view. At this granularity level, we do not provide any information on the behavior of O during its execution. So basically the execution can be symbolized by $O \wedge pre \Rightarrow O \wedge O_post$ as in figure 2.

In this context, we must introduce time, causality and epistemic expression tools but this is not the purpose of the current paper. The previous basic statement can be extended to express the semantic and temporal compatibility of an interaction by using the five following equations:

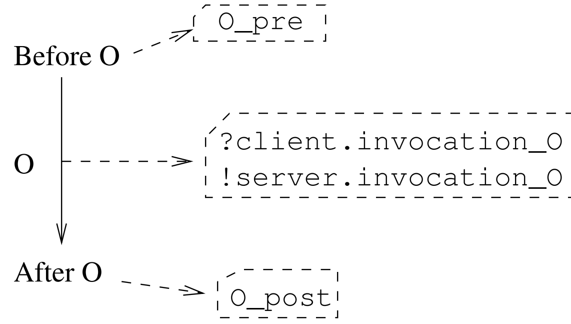
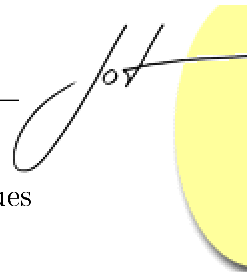


Figure 2: Temporal Logic Representation of an operation execution

1. $RO \wedge RO_pre \Rightarrow XO \wedge RO_post$
2. $PO \wedge PO_pre \Rightarrow XO \wedge PO_post$
3. $RO = PO \wedge XRO = XPO$
4. $RO \wedge RO_pre \models PO \wedge PO_pre$
5. $XPO \wedge PO_post \models XRO \wedge RO_post$

The first condition (1) tells that there exists a specification S_{RO} of a RO which enables the client to move from one coherent state to another. The second expression (2) states the existence of a provided operation specification (S_{PO}) which enables the server to pass from one coherent state to another. The three last conditions seal the relation of compatibility between the client and the server. The third condition (3) guarantees that the provided operation PO can be used instead of the required operation RO. In other words, the component can make a direct invocation without the need of a connector or an adaptator to link the PO and the RO. The fourth condition (4) imposes the contra-variance between the provided and required pre-conditions. The fifth condition (5) imposes the covariance between the provided and required post-conditions. In other words, before an invocation, each pre-condition from the client must be verified by the caller and at the return, each post-condition from the server must be in agreement with the properties of the called.

The pragmatic compatibility is the most difficult to define. It consists in the behavioural and environmental compatibility verification. Pragmatic properties are often difficult to analyse. They could be expressed totally by logical expressions but also as state-transition diagram. In this case specifying compatibility is equivalent to testing if the diagram of the PO is less constrained than the diagram of the RO. Typically, this is the notion of symmetries found in Petri nets. This type of compatibility will be transitive but the symmetry will not be guaranteed as it was in the other levels. A more complex definition of the compatibility can be defined for



interfaces, which will impose a causal order between calls. However the techniques are exactly the same than those used to compare the method behavior.

Semantic and pragmatic constraints can be expressed with first order predicate logic[2], second order modal logic[3] and with different languages [19]. Our study does not limit itself to one of them but all services are to be specified using the same logic language. It is not our purpose to develop our own language so we check some of among the abundance of existing ones. Our preference goes to one of the most widely used: OCL. In addition, temporal extensions and some software checkers are available for free [12]. Unfortunately, a knowledge expression extension is still missing in OCL. This add-on could be very useful to formulate and manage the ways and means of the information diffusion and knowledge of a specific data among components. A possibility to deal with theses properties is to use a runtime checker, to test if a component bind itself with an unauthorized entity or/and get access to forbidden data.

Interaction contracts

One of the essential principles in an interaction contract elaboration is the compatibility between the properties of each contributor. The compatibility between both involved operation is a sufficient and necessary condition for the existence of a contract (Figure 3) because of the transitivity of the relation.

So we must have $S_{RO} <: S_{PO}$. If this conditions is not verified then the interaction and subsequently the composition is not possible. An interaction contract has a specification issued from a negotiation process between a client and a server on the base of the required and offered specifications. So a simple interaction contract is a specification S_{CO} which uniquely types an interaction. The contract specification S_{CO} replaces, in the application using the caller and the called, their respective specifications (S_{PO} and S_{RO}). The existence of the S_{CO} is ensured by the relation transitivity. So we have:

- $PO <: CO <: RO$
- $RO_pre \models CO_pre \models PO_pre$ (contravariance)
- $RO \wedge RO_post \models CO_post \models PO \wedge PO_post$ (covariance)

[2]When the subject of the sentence is an individual object (like Socrates in "Socrates is mortal"), then we are using first order logic

[3]When the subject is another predicate (like being mortal in "Being mortal is tragic"), then we are using second order logic or higher order logic

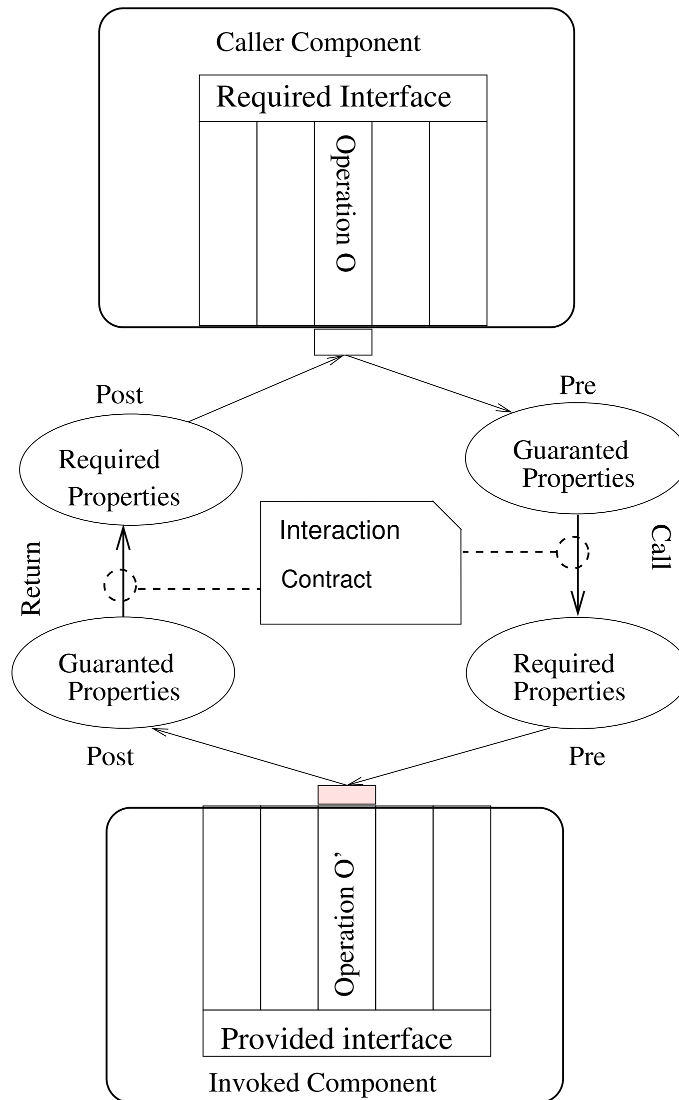
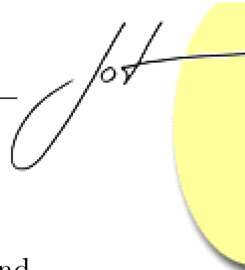


Figure 3: Representation of interaction contracts



A first trivial illustrative sample

Firstly, let us say that more informations and a more complex sample can be found on the project page of ACCORD [4]. But, to illustrate this, let us take a naive *getCash* operation, which withdraws cash from a bank account. The client coordinator has a required interface *RIBank* link with the bank component obtained through a provided interface *PIBank* from a bank component. The specified method is *getCash*. The Required Operation of the client component is:

```
Component Coordinator - interface RIBank
  OperationgetCash (amount, account)
  pre-condition (OR_pre) :
    typeOf (amount) is type_amount
    ∧ typeOf (account) is type_account
    ∧ amount < maximum1
    ∧ balance (account) - amount > 0
  post-condition (OR_post) :
    balance (account) = balance (account)@pre - amount
```

To be more precise, it would have been interesting to introduce the epistemic modalities where the client would only know if the operation is possible or not without knowing the balance. However, the previous code only limits the amount of money a client can take to a maximum of *maximum1* and imposes that the balance of the account will stay positive after the operation.

The specification S_{PO} of the provided operation (PO) would be:

```
Component Bank - interface PIBank
  Provided Operation :  getCash (amount, account)
    pre-condition (PO_pre) :
      typeOf (amount) is type_amount
      ∧ typeOf (account) is type_account
      ∧ amount < maximum2
      ∧ balance (account) amount > 0
post-condition (PO_post) :
      balance (account)=balance (account)@pre - amount
      ∧ balance (account) < triggerLevel(account) ==> msg_alert
```

This specification only enables *getCash* operation with an amount lower than *maximum2*. The post-condition throws an alert message if the amount on the acount is below a trigger value. In this case, the contra-variance hypothesis on the pre-conditions $RO_pre \Rightarrow PO_pre$) enforces the following condition for the operation to be successful:

```
amount<maximum1 ==> amount<maximum2
```

The maximum amount required must be less than the maximum amount provided. The covariance hypothesis on the post-conditions $PO_post \Rightarrow RO_post$) are naturally satisfied because

[4]http://www.infres.enst.fr/projets/accord/lot1/index.html

```
  [
    balance (account)=balance (account)@pre-amount
    ∧ balance (account) < triggerLevel(account) ==> msg_alert
  ]
  ⟹ balance (account) = balance (account)@pre - amount
```

So from these conditions, we can make a S_{OC} which is compatible with S_{RO} and S_{PO} :

```
Contract Operation:  getCash (amount, account)
  pre-condition (CO_pre) :
    typeOf (amount) is type_amount
    ∧ typeOf (account) is type_account
    ∧ amount < maximumC
    ∧ maximumC < maximum2
    ∧ balance (account) amount > 0
  post-condition (CO_post) :
    [
      balance (account)=balance (account)@pre-amount
      ∧ balance (account) < triggerLevel(account) ==> msg_alert
    ]
    ==> balance (account) = balance (account)@pre - amount
```

In S_{CO}, we must adopt in the pre-condition with an adapted maximum called *maximumC* which must be less than *maximum2* and in the post-conditions those from the client which are compatible but which enforce some more conditions.

3 COMPONENTS ASSEMBLY AND CONTRACTUAL DEPENDENCIES

Why interactions contracts are not sufficient

Some specific cases appear in the construction of the specifications of S_{RO} or S_{PO}. On one hand the programmer can use a "Component Off The Shell" along with the provided specifications defined by the "COTS". Henceforth, the developer accepts the post-conditions PO_post and makes them his own RO_post for the new implemented component. In the same manner, he uses the pre-conditions PO_pre for the RO_pre and tries to satisfy them. Therefore, the newly created component accepts all the conditions imposed by the specifications of the component with which it interact. Verifying the peer to peer compatibility between both elements is then trivial. This case often rises in top-down development methodologies. On the other hand, when the programmer develops a new component, he can use the specifications of a required operation to build a new component with a provided operation that will totally satisfy the required properties. This is the case of bottom-up methodologies.

But this is insufficient. A composite is composed of a set of components according to an assembly graph. As a component can be a composite itself, the abstract model is fractal. If there is no dependence between the specifications then the assembly is reduced to a succession of interaction contracts definitions. More often, there are dependencies between the specifications so the validity of the aggregation must be

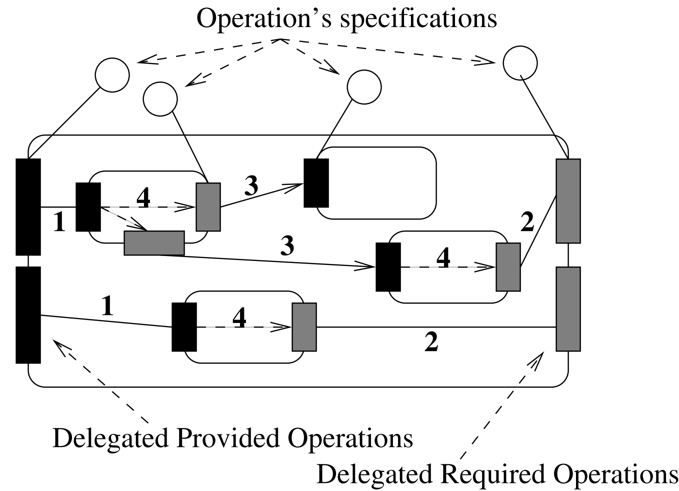


Figure 4: Structure of a composite

verified.

The component specifications are fixed and are not mutable at execution time. The Provided Service has a visible (public) part and a hidden (private) part with its respective specifications noted S_{VPO} and S_{HPO}. The Required Service also has a visible and hidden part respectively noted S_{VRO} and S_{HRO}. (figure 4).

Dependencies taxonomy

This model exhibits four categories of possible dependencies between components specifications (noted from 1 to 4 on figure 4):

1. Type 1 identifies the link between a visible provided interface of the composite and an interface provided by member components of the composite.

2. Type 2 identifies the link between the required visible interface of a member of the composite and a visible interface of it.

3. Type 3 identifies the external links between components. They are the previously defined links between a required operation and a provided one.

4. Type 4 identifies intra-components links. These are reified to support the dependencies created by the implantation of a service that are used by another one.

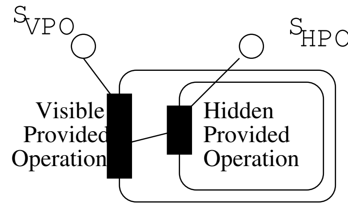


Figure 5: Dependencies of provided public services

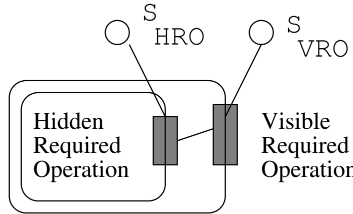


Figure 6: Dependencies of required public services

Dependencies of provided services (type 1 links)

Links of type 1 hold a more or less complex delegation of an external visible provided interface that is to say an invocation of a service on the external visible part of a component can be transferred to a sub-element of itself. The selected hypothesis is the pure delegation. In this case, the provided specification S_{VPO} that is rendered visible is the same as the specification of a provided service (S_{HPO}) of a composite member. Another working hypothesis would be a delegation by compatibility. Then a specification of a visible provided service is compatible with the provided service of a member of a composite ($S_{HPO} <: S_{VPO}$). Another last solution would be a delegation by adaptation so that the provided visible service is an adaptation of an internal provided one (figure 5).

Dependencies of required service (type 2 links)

As previously mentioned, one solution is the pure delegation of a specified required visible service of a member component as a required visible service of the composite. And in a similar way to the type 1 link, we can use the compatibility delegation ($S_{HPO} <: S_{VRO}$) and adaptation delegation.

Dependencies between components (type 3 links)

In the context of an assembly, a component needs a required service RO according to a S_{RO} specification. It also provides a service PO in agreement with to the specification S_{PO}. In the general case, a dependency relation called RPD (Required

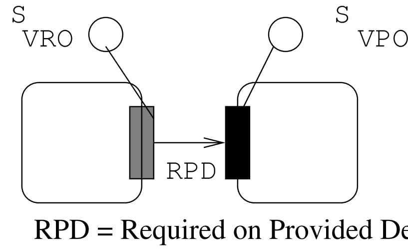


Figure 7: Inter composites dependencies

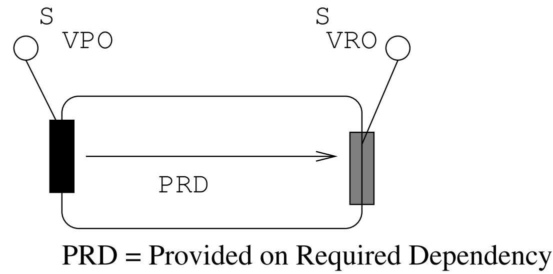


Figure 8: Internal Dependencies of a composite

on Provided Dependency) express that S_{RO} rely on S_{PO} by the relation $S_{RO} = RPD(S_{PO})$. Examples of such dependencies can be exhibited in static or dynamic links or at a larger scale in aspect oriented programming. With the static link, the copying mechanism of the provided specification to define the required specification is one sample of dependency. With a dynamic link a component receives a reference to another one at runtime. So, it depends only on the component that will finally really be invoked. Aspect Oriented Programming (AOP) [4] introduces an aspect component that require the service of a base component which is, at first sight, undefined.

The specification and the services for this aspect oriented component is the one offered by the wrapped component. This would enable AOP advantages of the high granularity level of component. A last case is when quality of service is used because it also implies numerous dependencies. For instance, the response time for a service relies on the final server and its intermediates. Then time overhead, parameter modification and so on can be introduced (Figure 7).

Internal Dependencies (type 4 links)

A composite provides a service PO according to a specification S_{PO}. It requires a service RO with its specification S_{RO}. S_{RO} relies on S_{PO} by a dependency function called Provided on Required Dependency noted PRD so that $S_{PO} = PRD(S_{RO})$. We have the same example as those previously quoted (figure 8).

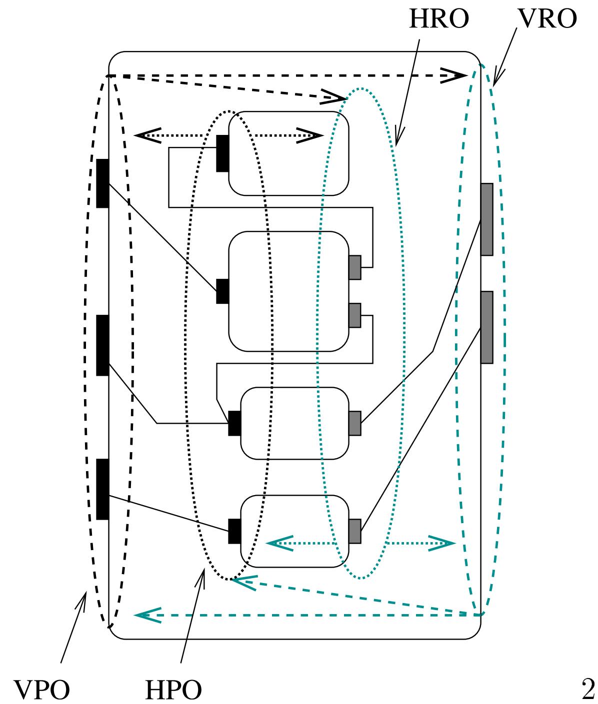


Figure 9: Internal structure and dependencies of a composite

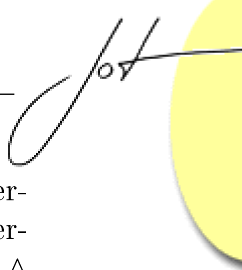
So finally we can structurally define a composite with the schema of the figure 9.

The dependency equations system

The observation of the structure (figure 9) leads to distinguish four set of services:

- a set of private provided services S_{HPO} that are hidden and provided by a member component and can only be internally accessed by the composite.
- a set of private required services S_{HRO} that are hidden and must be offered by a member component (or a set of them) to the currently studied composite.
- a set of public provided services S_{VPO} that are provided by the component to the world
- a set of public required services S_{VRO} that are required by the component. This link is often employed for shared public components.

From this list, we can now define the full equations for a typical component of the system.



- Internal-External component dependency enforces that a private provided service (S_{HPO}) rely on public required services (S_{VRO}) and private required services (S_{HRO}) according to a function PRD_H so that $S_{HPO} = PRD_H\,(S_{VRO} \wedge S_{HRO})$.
- Internal-External component dependency enforces that a public provided service (S_{VPO}) rely on public required services (S_{VRO}) and private required services (S_{HRO}) according to a function PRD_V so that $S_{VPO} = PRD_V\,(S_{VRO} \wedge S_{HRO})$.
- External-Internal component dependency enforces that a private required service (S_{HPO}) rely on public provided services (S_{VPO}) and private provided services (S_{HPO}) according to a function RPD_H so that $S_{HRO} = RPD_H\,(S_{VPO} \wedge S_{HPO})$.
- External-Internal component dependency enforces that a public required service (S_{VRO}) rely on public provided services (S_{VPO}) and private provided services (S_{HPO}) according to a function RPD_V so that $S_{VRO} = RPD_V(S_{VPO} \wedge S_{HPO})$.

A fixed-point solution

Defining a contract for a component leads to determine a unique specification S_{HCO} which will replace the private offered (S_{HPO}) and required (S_{HRO}) specifications for each assembly link. Such a contract exists only if the compatibilities constraints settle incompatible dependencies between offered and required services (see paragraph "Interaction contracts"): $S_{HPO} <: S_{HCO} <: S_{HRO}$. In fact, if the S_{HCO} exists it can replace all the private specifications (S_{HPO} , S_{HRO}) inside the equations so we should have :

$$\wedge \begin{cases} S_{HCO} = RPD_H(S_{VPO} \wedge S_{HCO}) \\ S_{HCO} = PRD_H(S_{VRO} \wedge S_{HCO}) \\ S_{VRO} = RPD_V(S_{VPO} \wedge S_{HCO}) \\ S_{VPO} = PRD_V(S_{VRO} \wedge S_{HCO}) \\ \quad S_{HPO} <: S_{HCO} <: S_{HRO} \end{cases}$$

This expression of the assembly, if it exists, relies on the contract specification defined for the private services (S_{HCO}) and the required and provided services (S_{VPO},S_{VRO}).

"No solution" traps

Until now, we supposed a good system design which enables an internal contract replacing all the contracts of the links inside the composite. To achieve this, we use the

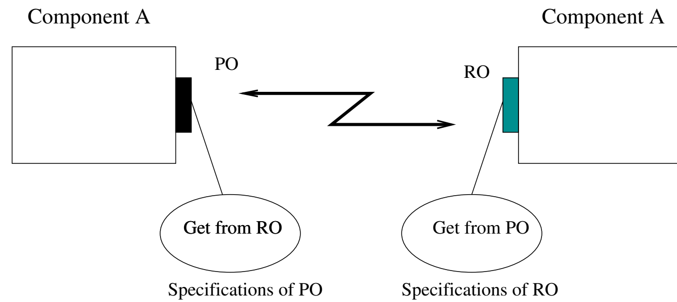


Figure 10: Cyclic relationship

peer-to-peer compatibilities and delegations. However, some errors could appear in the design so that the algorithm gives no solution. Continuing the analogy between our study and the mathematical fixed-point method, two types of incompatibilities can at least be identified. The first one is the non-convergence of the method and the other one is the detection of a direct incompatibility.

The verification of the assembly may not be completely specified. This is due to the system under-specification or to a cyclic delegation that cannot be resolved. This is shown in figure 10. In this context, the method could help the conceptor to isolate the defective components, contracts or interactions.

The direct incompatibility is relatively easy to isolate because it is between two elements. A simple example would be an interaction between two semantically different interfaces; one relying on a transaction and another one relying on an atomic operation. The only solution to the problem is the insertion of a connector to do "the glue job". But this direct incompatibility can interfere at different levels.

The first and most simple case concern a required service and a provided one. There are many solutions to reconcile two participants. Mediator, Aspect Oriented Programming, Connector are some of them. The second one can arise in the case of delegations that are not pure; that is to say when a service is delegated in compatibility. A delegation is a transfer of service from one component to the other. It can be viewed as a structural interaction or as a cooperation. Two objects cooperate to achieve a common aim. This type of interaction is often not reified by the system. If an incompatibility arise, a wrapper with some kind of code must be used to reconcile the delegate and the delegated.

4 A MORE COMPLEX (YET SIMPLE) EXAMPLE

Context

We set an academic sample for our exposé. A composite Cash Dispenser manages the distribution of money. It encapsulates some internal composites according to

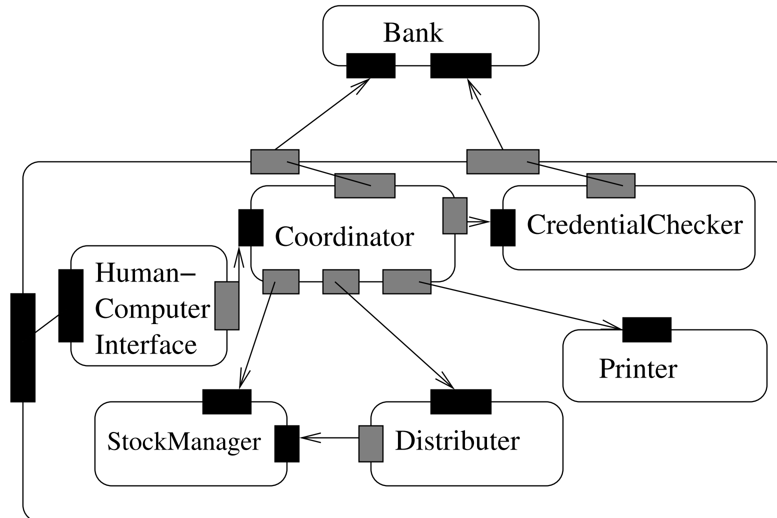


Figure 11: Global view of the system

the structure presented in figure 11 .

The sub-elements of the main composite are the Human-Computer Interface, the Credential Verifier, the Stock Manager, the Printer and a Coordinator. The main composite is in external relation with a bank component for money transactions. To illustrate our point of view, we will partly describe the assembly of all these entities. We will limit ourselves to the relationship between the Coordinator and the other components. To keep the whole thing short, only semantic logic predicates will be used. Pragmatic properties will be excluded. From now, we will apply a bottom-up approach. The essential steps are:

1. Verify the provided and required specifications compatibility.
2. Solve the fixed point equations to get the provided specifications and the contracts.

Coordinator-StockManager relationship

The role of the coordinator component is to control the system tasks. The only operation that will be studied is the method *getCash* from the Coordinator. The behaviour of this operation is described on figure 14.

For a cash withdrawal, the *Coordinator* begins by checking the available stock. Consequently, a Required on Provided Dependency exists between the components *Coordinator* and *StockManager* (figure 12).

It is characterized, on the *Coordinator* side, by the existence of a required interface called "*RIStock*". A basic specification expressed by pre and post-conditions is :

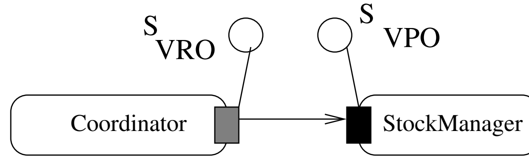


Figure 12: Coordinator-StockManager relationship

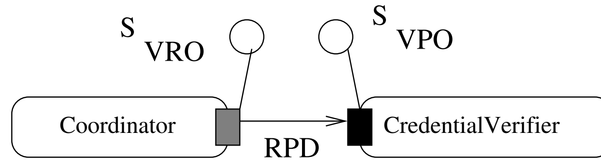


Figure 13: Coordinator-CredentialVerifier relationship

```
Component Coordinator - interface RIStock
  method:  check_stock (amount, notEnoughStock)
    pre-conditions:
      typeOf(amount) is amountType
      ∧
      0<amount<maxStock
    post-conditions:
      Stock.PIStock.check_stock.post
```

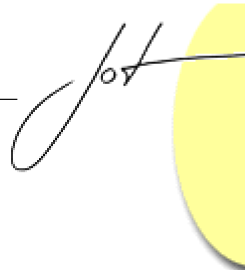
The requirer commits itself to satisfy typing and ceiling constraints from its pre-conditions. As said before, the requirer expects the provider to fulfil conditions in return. To demonstrate the dependency, the post-condition has been set in order for the required interface of the Coordinator ("*RIStock*") to be the one offered by the *StockManager* component: *PIStock*. This is the reason of the "*Stock.PIStock.check_stock.post*"line. The *StockManager* on its own sets up a self-sufficient provided interface called *PIStock*. It is specified by the following properties:

```
Component StockManager - interface PIStock
  method:  check_stock(amount, notEnoughStock)
    pre-conditions :
      typeOf(amount) is amountType
      ∧
      0<amount
    post-conditions:
      notEnoughStock=(amount>StockAmount)
```

Coordinator-CredentialVerifier relationship

The coordinator then proceeds to the credential check of the client. To do this, it uses the authenticate component method (see figure 13).

The Required on Provided Dependency is featured from the *Coordinator* side by



a *RIAuthenticate* :

```
Component Coordinator - interface RIAuthenticate
 method:  authenticate (userIdentity,password,invalidPassword)
   pre-conditions:
     CrendentialVerifier.PIAuthenticate.authenticate.pre
   post-conditions:
     CrendentialVerifier.PIAuthenticate.authenticate.post
```

And from the *CredentialVerifier* point of view, the *PIAuthenticate* is :

```
Component CredentialVerifier - interface PIAuthentificate
   method:  authentificate (userIdentity,password,invalidPassword)
     pre-conditions:
       typeOf(userIdentity) is loginType
       ∧
       typeOf(password) is passwordType
     post-conditions:
       invalidPassword=( passwordsFile(userIdentity)≠password)
```

The required interface does not bind any peculiar condition and accept those from the provided interface.

Bank-Coordinator relationship

The coordinator has a required interface *RIBank* link with the bank component obtained through a provided interface *PIBank*. The specified method is *withdraw_account*.

```
Component Coordinator - interface RIBank
 method:  withdrawAccount (userIdentity, amount, overdraft)
   pre-conditions:
     Bank.PIBank.withdrawAccount.pre
   post-conditions:
     Bank.PIBank.withdrawAccount.post
```

and,

```
 Component Bank - interface PIBank
   method:  withdrawAccount (userIdentity, amount, overdraft)
     pre-conditions:
       typeOf (userIdentity) is LoginType
       ∧
       typeOf(amount) is amountType
       ∧
       amount<maximumWithdrawal
     post-conditions:
       overdraft=(balance(userIdentity)<amount)
       ∧
       balance(userIdentity)=balance(userIdentity)@pre - amount
```

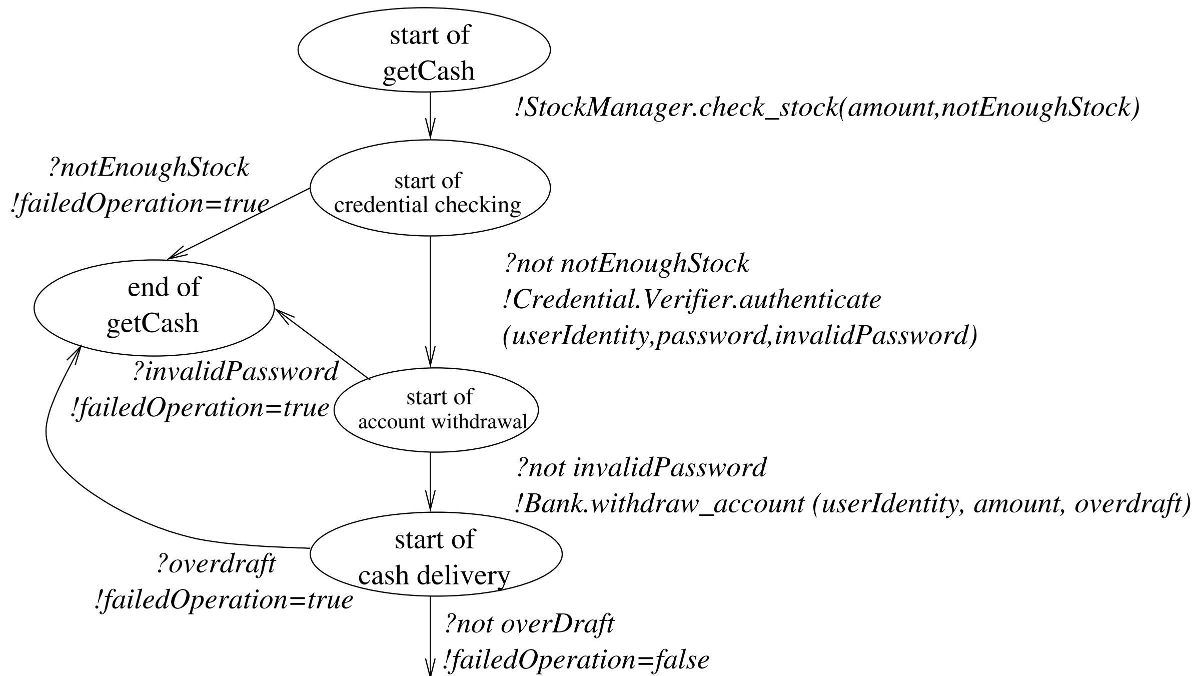


Figure 14: Internal Coordinator dependencies

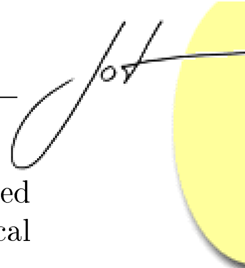
Internal dependencies from the coordinator component

The simplified version of the operation *getCash* on figure 14 gives us the main interface *PIMain* of the coordinator:

```
Component Coordinator - interface PIMain
  method:getCash (amount,userIdentity,password,failedOperation)
    pre-conditions:
      Coordinator.RIStock.check_stock.pre
      ∧ Coordinator.RIAuthentificate.authentificate.pre
      ∧ Coordinateur.RIBank.withdraw_account.pre
      ∧ ...
    post-conditions:
      Coordinator.RIStock.check_stock.post
      ∧ Coordinator.RIAuthentificate.authentificate.post
      ∧ Coordinateur.RIBank.withdraw_account.post
      ∧ ...
      ∧ failedOperation = (notEnoughStock ∨ invalidPassword ∨ overdraft)
```

Solving the fixed point equations

It is trivial to establish the external compatibility between the provided and required specifications of the encapsulated components. The required properties of the *Coordinator* are mostly defined by importing the provided properties of the used services.



Solving the equations is equivalent to substitute the specifications (also presented in this paper as split contracts) for the interaction contracts and a global logical reduction of them in the *Coordinator* component.

After the dependencies substitution, we get:

```
  interface PIMain
    method:getCash (amount,userIdentity,password,failedOperation)
      pre-conditions:
        typeOf(amount) is amountType
        ∧ 0<amount<maxStock
        ∧typeOf(userIdentity) is loginType
        ∧ typeOf(password) is passwordType
        ∧ typeOf (userIdentity) is LoginType
        ∧ typeOf(amount) is amountType
        ∧ amount<maximumWithdrawal
        ∧ ...
post-conditions:
        ∧ notEnoughAmount=(amount>StockAmount)
        ∧ notValidPassword=(passwordsFile(userIdentity)≠password)
        ∧ overdraft=(balance(userIdentity)<amount)
        ∧ balance(userIdentity)=balance(userIdentity)@pre-amount
        ∧ ...
        ∧ failedOperation = (notEnoughStock∨invalidPassword∨overdraft)
```

And after the logical reduction:

```
  interface PIMain
    method:getCash (amount,userIdentity,password,failedOperation)
      pre-conditions:
        typeOf(amount) is amountType
        ∧ 0<amount< min(maxStock, maximumWithdrawal)
        ∧ typeOf(password) is passwordType
        ∧ typeOf (userIdentity) is LoginType
        ∧ ...
      post-conditions:
        balance(userIdentity)=balance(userIdentity)@pre-amount
        ∧ failedOperation =
        [
          amount>StockAmount
          ∨ passwordsFile(userIdentity)≠ password
          ∨ balance(userIdentity)<amount
        ]
        ...
```

Finally, a specification of the Coordinator provided interface *getCash* has been found. The next step in the bottom-up approach is to proceed to the resolution of the “Human-Computer interface” component contracts.

5 CONCLUSION

In this paper, we present a new component assembly approach based on a fixed point equation resolution. This method can be applied to a top-down and bottom-up accurate existing components aggregation. One difficulty is the lack of specification for required and provided services of industrial components. The syntactic signature is often the only specification. User manuals are not really usable to make formal verifications. Moreover, even when these specifications exist, manual or automatic verification is still difficult. Split expression provided and required specifications, when assembled, insure correctness in component based applications.

It also enables to have a better isolation of each component, a new easier way to bind them and a possibility to perform structural verification of application. As a future work, we intend to design a method to locate assembly flaws hence to correct a bad assembly.

References

[1] A.Thomas. Enterprise javabeans technology. White paper, Sun Microsystems, Inc., 1999.

[2] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, june 1999.

[3] D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, http://www.catalysis.org, 1998.

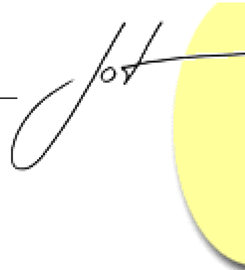
[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *LNCS 1241*, November 1997.

[5] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proc. of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems*, pages 169–180, Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada , 1990.

[6] J.-M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts.* Addison-Wesley, October 1999.

[7] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[8] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.



[9] B. Meyer. *Eiffel : The language*. Prentice-Hall, 1991.

[10] B. Meyer. Applying Design by Contract. *Computer*, October 1992.

[11] Microsoft. .NET. *http://www.microsoft.com/net/default.asp*, 2001.

[12] Klasse Objecten. OCL Checker. Web site, Klasse Objecten, 1999.

[13] Object Managment Group. Corba components - joint revised submission. Technical report, Object Managment Group, March 1999.

[14] Parasoft. jContract. Web site, Parasoft, 1996.

[15] P. Champagnoux, L. Duchien, D. Enselme, G. Florin. Typage pour des composants coopératifs. In *NOTERE 2000 Proceedings*, November 2000.

[16] R.Marvie. Corba components: la proposition unifiée, du modèle au objet au modèle des composants. Technical report, LIFL, May 1999.

[17] R. Fagin,J. Y.Halpern, Y. Moses, M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[18] R. Kramer. iContract - The Java Design by Contract Tool. Web site, iContract, 1999.

[19] S. Shapiro. Second order logic, foundations, and rules. *Journal of Philosophy 87, pages 234-261*, 1990.

ABOUT THE AUTHORS

Fabrice Legond-Aubry is a PhD student at CNAM ("*Conservatoire National des Arts et Métiers*", Paris, France) since 2001. He is currently working with the University Paris VI LIP6 laboratory. He is involved in the development of the JAC core framework (Java Aspect Components). He also works on components adaptability and re-usability by using aspects and contracts. In 2000, he graduated from the EFREI ("*Ecole Francaise d'Electronique et d'Informatique*") engineering school.

Daniel Enselme is Assistant Professor in Computer Science at CNAM. He received the PhD from the University of Paris VI in 1985. He has been working on Natural Language Processing and is now interested in Component Based Software and Aspect Oriented Programming.

Gérard Florin is currently a University Professor in computer science at CNAM. He received the "*doctorat d'état*" degree from the University of Paris VI in 1985. His research interests included Stochastic Petri Nets, Distributed Computing and now software components and Aspect Oriented Programming.