# Adapting Software Components by Structure Fragmentation

Gautier Bastide
Ecole de Mines de Douai
941 rue Charles Bourseul
59558 Douai, France
bastide@ensm-douai.fr

Abdelhak Seriai
Ecole de Mines de Douai
941 rue Charles Bourseul
59558 Douai, France
seriai@ensm-douai.fr

Mourad Oussalah
LINA, université de Nantes
2 rue de la Houssinière
44322 Nantes, France
oussalah@lina.univ-nantes.fr

## ABSTRACT

We present in this paper an approach aiming at adapting software components. It focuses on adapting component structures instead of adapting component services. Among the motivations of this kind of adaptation, we note its possible application to permit flexible deployment of software components and flexible loading of component code according to the available resources (CPU, memory). Our adaptation process is based on the analysis and the instrumentation of component codes. It respects the black-box property when it is implemented as a service provided by the component to be adapted. To support this structural adaptation technique, we developed an adaptation process which we have experimented using the Java framework of the Fractal component model.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

## General Terms

Design, Languages

## Keywords

Software component, structural adaptation, object-oriented, refactoring

## 1. INTRODUCTION

Component-Based Software Engineering (CBSE) aims at building applications by assembling reusable components [24]. However, experiments show that direct component reuse is extremely hard and this one usually has to be adapted in order to be integrated into an application. This difficulty is due to the available large variety of infrastructures and software environments, going from the simple mobile phone equipped with minimal capacities to the cluster of several multi-processor computers. The solution may consist in the development of software components which can adapt themselves to these constraints. To deal with this issue, many approaches were proposed to adapt component-based applications. They differ according to several criteria. Among these last, we can quote: the adaptation target, the adaptation moment, the adaptation actor, the adaptation goal, etc. For example, the adaptation target can be the application architecture [19], the components [6] or the component connections (i.e. connectors) [4]. Concerning the adaptation moment, an application can be adapted during its conception, its deployment [9], [20] or its runtime [19].

Nevertheless, we can note that, in spite of this diversity of proposed approaches, all ones which are interested to adapt components focus on adapting their services and only some works are interested to adapt the component structures. Moreover, to our knowledge, all these last approaches are interested in the adaptation of the component implementation by the replacement of an algorithm by another (i.e. code replacement). The result is that, no approach proposes techniques for restructuring software components (i.e. to adapt their structures). The component restructuring can concern either component external structure (e.g. ports, interfaces or services) or component internal structure ( e.g. its sub-components, its internal connexions).

While being based on the above considerations, our objective in this paper is to propose a software component restructuring approach (i.e. allowing adaptation of the component structure). In fact, our approach permits to restructure software component implementation (i.e. internal structure). We interested to components based on an object-oriented implementation and we experiment this approach using Fractal component model [8] and its Java implementation named Julia [7].

We discuss the proposed approach in the rest of this paper as follows. Section 2 presents the general mean of the structural adaptation of a software component then describes our study focus and an example of experimentation used to illustrate our approach. Section 3 describes adaptation process and how to guarantee component coherence and communication. In section 4, some related works are briefly mentionned. Finally, conclusion and future works are outlined in section 5.


*SAC'06* April 23-27, 2006, Dijon, France

## 2. RESTRUCTURING SOFTWARE COMPONENTS

### 2.1 General presentation and motivations

The structural adaptation of a software component consists in the capacity to modify the structure of this one while preserving its behavior and its services. In fact, the structural adaptation is interested neither to modify the behavior of the component nor its services, but rather to reorganize its structure. Indeed, the structure of the component which has to be adapted consists of the list of the structural elements, defined in this component, as well as their links. For example, a component can be seen as collections of ports, interfaces, sub-components, classes, etc. Thus, the structural adaptation can consist of updating the list of a component interfaces by modifying the content of each of these interfaces or restructuring the content of the component ports. Also, the structural adaptation can consist in the decomposition of a software component in some sub-components where the set of provided services must be the same as that provided by the initial component before its adaptation.

Adapting the structure of a software component can be useful in many cases. For example, external structural adaptation can be use in order to increase component reuse and internal structural adaptation for a flexible component deployment.

First, component restructuring can be use in order to match heterogeneous component external structures. In fact, to assembly two components together or to integrate a component to an existing application needs to restructure their provided and required interfaces (or/and ports) if it proposes all required services but when these one are not structured in interfaces permitting to match them. For example, figure 1 shows that the C1 and C2 components can't be assembled because C1 required ports don't match with thoses provided by C2 although C2 provides all required services. So, using our approach for restructuring ports, matching between C1 and C2 becomes possible.

Moreover, the component structure adaptation by its fragmentation in some generated components can be used to define a flexible deployment strategy. Indeed, this adaptation permits to organize component services in separate sub-sets and defining each sub-set in a new component generated by fragmenting the original one. Then, the generated components can be deployed on one machine or separately on different ones which can be distributed or not. These possibilities are very useful for load balancing and to increasing performance (Fig. 1).

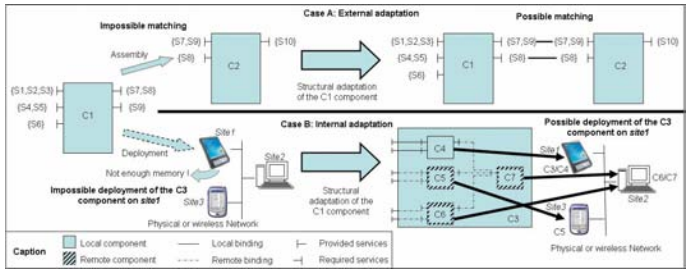


**Figure 1: Structural adaptation**

### 2.2 Our approach focus

Our approach is developed considering the following directions :

*Restructuring software components by fragmenting their object-oriented implementation*: Our approach focuses on allowing component internal structure fragmentation to fulfil some needs as deployment flexibility. We consider here an oriented-object implementation of components. Thus, the internal structure of a component consists of some class hierarchies representing the implementation code of its services and its interfaces. Our approach deals with the Fractal component model [8] and its Java implementation called Julia [7] developed by the INRIA[1]. Many deciding factors have motivated the choice of Fractal. In fact, this model is a hierarchical component model quite closed to UML2 model [11] with the only difference that it not define port concept. In this model, provided and required services are structured in interfaces which are used by components when they communicate with each other. Each server interface gives access to a set of operations. And, each client interface defines a set of operations that the component may invoke. To create a Fractal component, using the Java implementation, it necessary to specify a unique class which implements all services (i.e. methods) specified by the corresponding component interfaces.

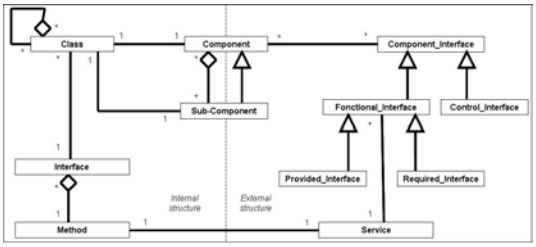


**Figure 2: Fractal component model**

*Static component restructuring*: The adaptation process is realized in static way and it needs component source code (i.e. static adaptation). In fact, component source code parsing, transformation and instrumentation are used in order to generate new component codes resulting by the fragmentation of the initial one. This process can be implemented in two ways. The first one preserves the black-box property of software component and the second one don't. The first technique consists of implementing adaptation process during the component design as an extra services provided by a non-functional interface of the component to be adapted. In this case, component code source is encapsulated[2] and can be handled only through this interface. The second technique to implement the adaptation process is to design it as a separate adaptation tool which need source code of the component to be adapted.

### 2.3 Example of experimentation: a shared-diary system

In order to illustrate our purpose, we use throughout this paper an example of a software component providing services

[1]The French National Institute for Research in Computer Science and Control. http://www.inria.fr/

[2]Source code can be handled using cryptic format.

of a shared-diary system accessible to multiple users. We have implemented this component using the Java platform of the Fractal model. The component restructuring is realized in order to adapt the component deployment following the existing infrastructures. Different component instances are deployed. Each instance is generated by intanciation of a copy of the component obtained by a specific component structure adaptation (i.e. specific code transformation) to match with the specific infrastructure (e.g. one machine, distributed machines or PDAs). The component to adapt is a monolithic component which provides the following services:

1. Managing user personal diary. This includes authentification, consulting events (e.g. meeting, activity, project), querying the diary, etc. These services are provided through the *Diary* interface.
2. Organizing a meeting. This includes services permitting to confirm the possibility to organize a meeting where the date and the list of the concerned persons are given as parameters, services returning possible dates to organize a meeting of some people, etc. These services are provided through the *Meeting* interface.
3. Managing absence. This includes services permitting to verify the possibility to add an absence event, to consult all the absence dates of one or some persons, etc. These services are provided through the *Absence* interface.
4. Right management. This includes services concerning absence right attribution, service related to diaries initialisation, etc. These services are provided through the *Right* interface.
5. Updating the diary, the meeting dates, the absence dates and the absence rights of a person. These services are provided, respectively, through *DiaryUpdate*, *MeetingUpdate*, *AbsenceUpdate* and *RightUpdate* interfaces.

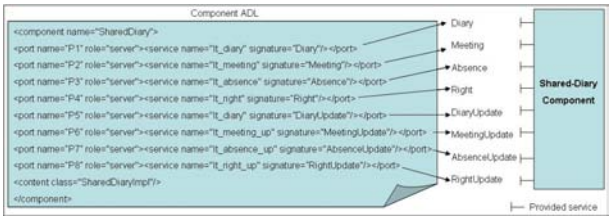


**Figure 3: Specification of the *shared-diary* component**

# 3. STRUCTURAL ADAPTATION OF SOFTWARE COMPONENTS

As we announced previously, the restructuring of a component aims at generating new components by the fragmentation of component based on an object-oriented implementation. This operation is realized without generating any change on the services provided by the component nor on its assembly with the others components of the application. To answer these constraints, we defined a restructuring process made up of four stages. First, we present these stages in section 3.1, then we focus on component coherence and communication problems in section 3.2.

## 3.1 Adaptation process

Component restructuring is achieved through a process composed of four stages (Fig. 4). The first stage is the specification of the needed new structure of the component to adapt. It aims at indicating which new components are to be generated by the fragmentation of a specified component. The second stage permits the component fragmentation according to the given specification and the generation of the needed new component structures. The third stage concerns the assembly of the new generated components and their coherence guarantee. It focuses on defining for each generated component its provided and required interfaces and to realize the adequate connexion between these component interfaces in order to preserve coherence of the related components. Finally, the last stage aims at integrating the generated components. It alllows to ensure that component adaptation is done without needing any change to do on other components to be connected to the component after its adaptation (i.e. of course if this is not needed before the component adaptation).

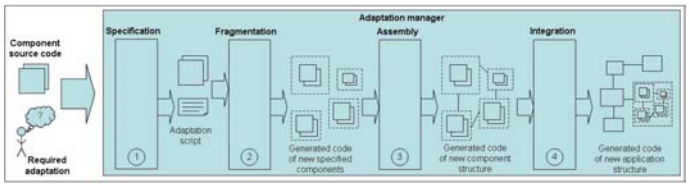


**Figure 4: Structural adaptation process**

### 3.1.1 Specification of the new component structure

The first stage of the adaptation process consists in the specification of the needed results. Its aim is to indicate which new components are to be adapted. The generated components are specified by designating their provided interfaces. In fact, these interfaces are considered as monolithic (i.e. unbreakable) elements. The specification is created through an adaptation script specified by using a notation close to XML [15] language[2] indicating for each new component its ports and for each port, the associated interfaces. Two generic directive formats, specifying what restructuring is to operate on a component called *CompToAdapt*, are given below. The first format corresponds to the case when the adaptation process is not implemented as a provided service of the component to adapt itself (i.e. white-box approach). The second format corresponds to the opposite case (i.e. black-box approach). In this last case, *AdaptInterface* and *StructAdat* respectively represents the adaptation interface and the service provided for the structural adaptation.

[2]Symbols " + ", " ∗ " indicate respectively one or more and zero or more elements. "{}" symbolizes a set of elements. When an interface is defined in several generated component, symbol "||" associated with the interface name indicates that this interface must be that which is used by the rest of the application.

```
StructuAdapt (CompToAdapt,
{CompDef = <{PortDef={[||] InterfaceDef}+ }+? >}*)

AdaptedComp.AdaptInterface.StructAdapt.(
{CompDef = <{PortDef = {[||] InterfaceDef}+}+>}*)
```

To illustrate this, let's reconsider our example of the shared-diary application, the goal of the structural adaptation is to reorganize services provided by this one in four new generated components (e.g. *Diary-Manager*, *DataBase-Manager*, *Absence-Manager*, *Meeting-Manager*). The script allowing to specify the needed structure is the following:

```
Shared-Diary. Adapt-Interface. Struct-Adapt (
 {Diary-Manager=<{P-Diary=Diary,DiaryUpdate}>}
 {DataBase-Manager=<{DB=Right, RightUpdate}>}
 {Absence-Manager=<{P-Absence=Absence,AbsenceUpdate}>}
 {Meeting-Manager=<{P-Meeting=Meeting,MeetingUpdate}>}
)
```

### 3.1.2 Component fragmentation and generation of the needed new component structures

Specification done during the previous stage is used in order to generate the implementation of each new component created by adaptation.
The component fragmentation aims at splitting component services and consequently the corresponding component internal implementation in some sub-collections of services where each sub-collection will be provided by a corresponding component. The definition of these components requires the generation of their external implementation in term of interfaces and services and the corresponding internal implementation in terms of object code. Two conditions are to be checked:
The first condition is the construction of each new component so as to guarantee its integrity. The integrity of generated components means that these ones function correctly without syntactic bugs. Our solution for that is a top-down construction of component to be generated. Starting from ports and interfaces specified for this component, all structural elements (i.e. ports, interfaces, methods, etc.) which are directly or indirectly dependent of these ones must be accessible within the component naming space.
The second condition consists in the coherence of the new generated component. In order to guarantee this, we propose a solution operating on two levels. First, at the component interface level, we introduce specific interfaces used to ensure communications between connected components (see Sect. 3.1.3). Second, at the object implementation level related to services of these interfaces, we instrument source code (see Sect. 3.2).
To guarantee the integrity of generated components, it is necessary to determine how structural elements composing a component can be dependent one to each other. In fact, structural elements can be dependent through two types of links: structural or behavioral links. For example, a port P1 composed of an interface I1 is structurally dependent of this last. However, if a method M1 calls a method M2 then M1 depends on M2 through a behavioral link.
Thus, all dependent structural elements, related to the definition of a generated component, form a graph which we called SBDG (Structural and Behavioral Dependence Graph). The SBDG is formed by analyzing, in top-down manner and starting from the specified component ports and interfaces, their source codes. Several behavioral links which appear on the SBDG aren't determinist because the polymorphism property of the object-oriented code. Thus, SBDG includes for a method all possible cases of dependencies considering polymorphism property. The nodes defined in a SBDG related to a generated component are the structural elements accessible within this component naming space. Each structural element of this graph must be defined in the corresponding component naming space.
In addition, to determine structural elements to be defined in a generated component naming space (i.e. its implementation), it is necessary to determine resources used by these elements. Resources can be internal or external. Internal ones are structural elements having states which can be updated and whose persistence is more important than that of the method in which it is used. When considering an object oriented implementation, internal resources are instances and class attributes. External resources are persistent elements as files and databases. We limited ourself here to consider only internal resources.
For example, figure 5 shows a part of the SBDG corresponding to *Meeting-Manager* and *Absence-Manager* generated components. As the *checking_meeting* method is linked to the *is_absent* method (i.e. the *checking_meeting* method of the *Meeting* interface calls the *is_absent* method of the *Absence* interface) which is contained in another interface, it is needed to create a behavioral link between the *Meeting* and *Absence* interfaces.

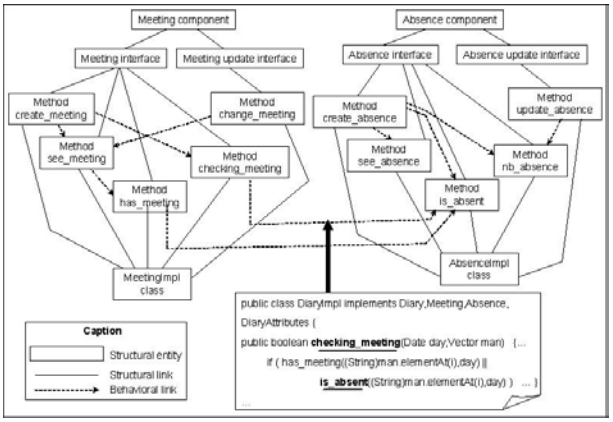


**Figure 5: A part of the structural and behavioral dependencies graph for the *Shared-diary* component**

### 3.1.3 Assembly of the new generated components

After the components specified at the time of the specification stage were generated, they have to be assembled. This assembly reflects links which exist between these components. These links are of two types. The first type of dependence relates to the behavioural dependence links between these components. A behavioural link reflects the use, by a generated component, of a given service defined by another generated component. The second type of link reflects the resource sharing between two generated components.
Behavioural dependances are setting-up through the implementation of services provided by specific interfaces added to the generated component structures: first, a required interface implemented for each component and which include all

needed services implemented by other components; second, a provided interface implementing all services implemented by this component and required by other components.
The shared-resource dependences are translated in the form of interfaces added to the structure of each component. Connexion of these interfaces allows the sharing and the communication of a common state of the shared resources. To materialize shared-resource dependences, we have defined for each generated component defining a shared-resource two interfaces:

- The first interface is defined as required and synchronous. It allows the component to notify to all components sharing a resource with it, its state modification. This interface is synchronous because the component which updates the resource state can continue its execution only after the other components which share this resource take into account this state modification. It allows to guarantee that components have a coherent state and consequently their services.
- The second interface, defined as provided, allows a component to receive the notifications of shared-resource updates.

Figure 6 shows an example of notification interfaces which are used in order to manage the resource *Absence_list*. This resource is an instance attribute whose value represents day of absence for a given person. It is shared by the components *Absence-Manage* and *Meeting-Manager*. Notification interfaces operate as follows: (1) When the resource *Absence_list* is updated by the component *Absence-Manager*, (2) a notification is send to the component *Meeting-Manager* . (3) Then, this last one memorizes the new value. (4) Once the value updated, it can manipulate this resource.

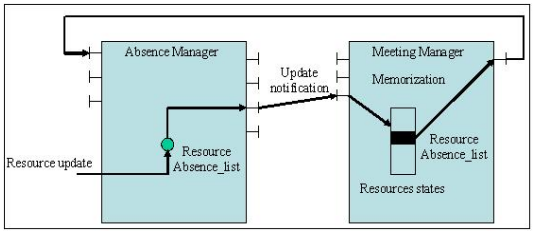


**Figure 6: Example of communication interfaces**

In addition to the notification problem, it is necessary to ensure that shared resource cannot be handled simultaneously. To prohibit simultaneous handling of these resources, we defined two additional interfaces:

- The first interface, defined as required and synchronous, allows it to get a resource access right. This interface guarantees that only one access to the resource is possible for the given time.
- The second interface defined as provided and synchronous allows a component to notify the availability of a resource to requester components.

Figure 7 shows an example of synchronization interfaces which is used in order to manage the *nb_day_free* resource. This resource is an instance attribute whose value represents the number of free days for a given person. It is shared by the *Absence-Manager*, *Database-Manager* and *Diary-Manager* components. First, *Absence-Manager* component which needs to update the *nb_day_free* resource (1) asks a right access to the other component which shared this resource (e.g. *DataBase-Manager* and *Diary-Manager*) (2). Then, after it receives a notification from these components, *Absence-Manager* can update the resource *nb_day_free* (3).

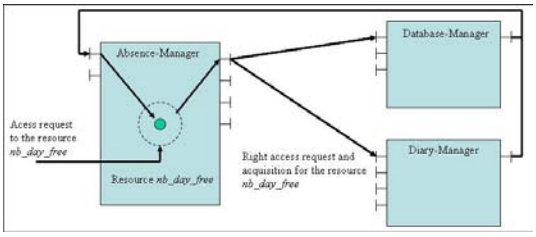


**Figure 7: Example of synchronization interfaces**

### 3.1.4 Integration of the structural adaptation result

The last stage of the adaptation process is the integration, in the subjacent application, of the structural adaptation result, which was obtained during the previous stages. It consists in connecting the new created components with the other application components and to guarantee that the component adaptation is achieved in a transparent way compared to the application components. In fact, the application must continue to run without any change compared to its initial configuration (i.e. configuration including the component before its adaptation). For that, it is necessary to satisfy the following properties:
First, the application shouldn't be able to access, after the adaptation, to other services than those which are provided by the component before its adaptation. This means that all the new services, being accessible to the new created components, must be inaccessible to the other components of the application. The new services are mainly linked to the management of the structural entity coherence and thus they don't must be accessible by an external component. Thus, the interfaces created for the realization of the adaptation process must be internal to the created components. They must be masked for all the external requests. For example, it would be forbidden to be able to access, by the application components except those created by adaptation, to the state of a shared resource.
Next, the new created (sub-) components could be accessible independently one from each other. For example, it would be possible to deploy separately these sub-components. It hasn't to provoke modifications on the rest of the application.
Our approach consists in encapsulating the new structure resulting from the adaptation in a new component (i.e. composite-component). A composite-component is a component which contains one or more other components called sub-components which can be primitive or composite. This new component (Fig. 8) allows to mask access to the component interfaces resulting from the adaptation (i.e. it can filter unauthorized accesses). Moreover, it is necessary to integrate to the composite-component some services used to handle its sub-components which are created after the adaptation. For ex-

ample, these services allow to define deployment procedures associated to each one of these sub-components.

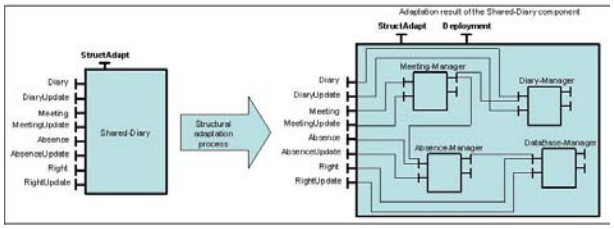


**Figure 8: Structural adaptation result for the shared-diary component**

## 3.2 Component coherence and communication

As we explained previously, the coherence of the generated components is ensured on two levels. Firstly, by considering the external structures of the generated components. This is done by the addition of new interfaces ensuring the communication between the components in question. Secondly, by the implementation of these communication interfaces. This is done, mainly, by the instrumentation of the source object-oriented code corresponding to these interface services.

### 3.2.1 Notification management after a resource update

After updating a resource in one generated component, the new state of this resource must be communicated to the other generated components. Thus, it is necessary to determine how and where this resource is updated within the component codes and next to instrument source code by adding notification instructions from one component to the others.

The first question is how a resource can be updated. This operation can be realized by using a direct or indirect reference to this one and by using an update instruction. In the case of Java, a resource (i.e. an attribute) can be updated by using it directly or using a reference to this one. The last case appears when we use an instance as an argument when calling a method [3].

Moreover, we need to distinguish update instructions from those which are not. For a primitive resource, this operation is easy because resource is updated using affectation instructions. This is more complicated when the resource is an object. This means that this resource can be updated using methods. For that reason, we instrument code by saving the resource state before its manipulation. This state is compared with that obtained after the instruction execution. If their states are different, a notification to the other components is needed. Other instrumentation instructions are related to this one. It is about methods needed to get resource states. These ones are added to the corresponding instantiation classes.

To illustrate this point, reconsider the shared-diary example. The class implementing this component defines two attributes named *absence_Dates_list* and *right_max* which represent respectively the absence dates for people considered by the shared-diary and the number of absence days which a person still has right. These attributes are considered as shared resources. In fact, the *add_new_absence_day* and *consult_remaining_days* methods use both these attributes. After the structure updating, these methods are defined respectively in two separate generated components: *Absence-Manager* and *Right-Manager*.

The last issue is about the moment of notification sending: when we have to send an update notification? In fact, the moment of this notification depends on the moment when components which receive the message take into account this notification. Indeed, it is useless to notify to these components several updates if it is only the last one which will be taken into account. Indeed, this proposal is true only if several resource update operations are gathered in the same critical section blocking for the other components. In this case, these last cannot modify the state of this resource which the execution stream of the component is at the end of this section. Thus, the operations of resource update notification are inserted at the end of each critical section (see below) which is defined for this resource.

### 3.2.2 Management of concurrent access

The resource access synchronization is the second condition to ensure the coherence of the generated components and thus the validity of the results which are returned by the execution of different services. The access synchronization consists in the ban of all simultaneous accesses to a same resource. The solution lies on the placement of access operations inside sections which are protected by semaphores (i.e. section of mutual exclusion).

#### 3.2.2.1 Critical section setting.

Critical sections (i.e. section of mutual exclusion) which allow to ban concurrent access are setting up through locks within entire services. So, we define a service critical section as the code fraction which includes all write or read resource accesses. In fact, locks are setting up between the first and last resource access instructions (i.e. first and last with regard to the entire service: linked to the initial component). We use the static code analysis in order to determine the possible exclusion section starts and ends which are defined on a resource then we instrument code according to these data. However, in order to favor parallelism, the critical section size has to be the shortest as possible. So, the effective set-up of an exclusion section is realized dynamically (i.e. during the service runtime). In fact, the critical section set-up requires two steps(Fig. 9):

First, we search all the possible entry points for the service critical section. For that purpose, it is necessary to parse the method source code in order to find resource access points. Then, code is instrumented before each resource access in order to allow service to put a lock dynamically (i.e. just before the first resource access). Indeed, during the application run, when a critical section starting point is detected, service checks the lock state. If it is not activated, then service has to start a critical section (i.e. to lock resource accesses). Else, nothing is done.

Second, we search all the possible critical section exit points. For that purpose, it is necessary to determine the resource updating points (see Sect. 3.2.1). We build the execution stream graph of the entire service. We extract a sub-graph whose nodes represent the update resource instructions or the first instructions which are executed just after a fork

[3]In Java, primitive attribute (e.g. integer, real numbers, char, boolean and string) cannot be called using a reference.

within the stream graph. Then, we analyze it in order to determine the critical section exit points. Our algorithm which is used for setting up the lock release tools is the following one:

```
For each path
    For every node N
        P = {path p / p starts from this node,
                      its end is an exit point and
                      it contains a resource access
                      point}
        If P=∅ Then
          add(N,List of possible critical section exit
          points);
        EndIf
    EndFor
EndFor
```

add(N,L): method which add the node N inside the list L.

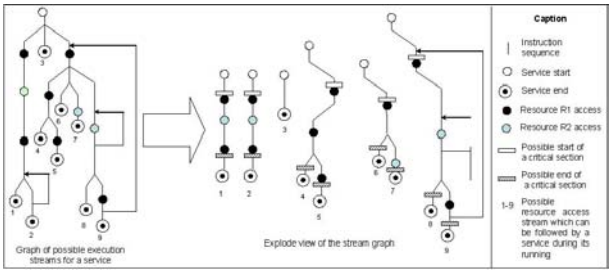


**Figure 9: Critical section setting**

#### 3.2.2.2 *Concurrent access to a shared-resource.*

The concurrent access to a shared-resource is carried out by the installation of mutual exclusion sections. These sections created by the instrumentation of the source code make it possible to manage the single access to a resource. The activation of a concurrent access within a section of code handling a resource is carried out by the request for this right. Concurrent access right is materialized by the acquisition of what we call "the token". Our approach requires the use of a stack for every component and a sequencer [3]. Each element of the stack contains an identifier of the service which asks token and the set of resources which are concerned. The stack elements are ordered according to their identifier.

The process for getting a token (i.e. for starting a critical section) is the following one: when a service wants to start a critical section, it asks the sequencer for an identifier which allows to order requests. Then, it puts this identifier and the set of needed resources into its stack. So, it sends its request to the other components by supplying these data (e.g. identifier and set of needed resources). When a component receives its request, it puts that in its stack according to the number which was supplied by the sequencer. Then, it sends back a delivery notification. A component gets the token only when there is no element of the stack before it which contain a resource included in its set and when it receives a delivery notification from all other components sharing the resource.

When the service which has the token aborts a critical section, this last one is deleted of the stack.

When the service calls another service, it gives the token and waits for. Thus, a service can need to send the token from a method to the other one. As two different methods can have the same signature, every method (i.e. service) has to know in which context they are called (i.e. by which method). Thus, methods have to send an identifier of their entire service.

For example, if there are two components C1 and C2 which provide respectively the services A and B, service A of the component C1 calls the service B of the component C2, it is necessary that the service B knows that it is called within the context of service A. Indeed, if service A locks the needed resources before calling the service B, when the service B is running, it must know that it has the token (i.e. in order to release the lock) because the service B is included in service A.

In order to ensure the token transfer, we introduce a new optional parameter for each method (i.e. service). Thus, it allows services to determine in which context it is called. When this parameter is indicated, the service is called within the context of a specified service. Otherwise, if this parameter is not indicated, method is the first which is called inside the entire service. However, one of the adaptation constraints is that we can't modify component interfaces because they can be used by other application components. Thus, new interfaces which contain the internal method signatures (i.e. signatures of services which are called by other) should be created.

## 4. RELATED WORKS

We classify related works according to two criteria. First, we presents works related to software component adaptation. The other criterion focuses on works which topic is restructuring program codes and more particular object-oriented ones.

If we consider the first criterion related to the work goal, many adaptation approaches have been discussed in the literature. Broadly speaking, adaptation techniques can be categorized as either white-box and black-box. White-box techniques typically require understanding of the internal implementation of the reused component, whereas black-box techniques only require knowledge about the component's interface. A commonly discussed black-box technique is wrapping, also known as containment in COM literature. Superimposition [5] is an alternative technique, the idea behind which is that the entire functionality of a component (i.e. rather than that of a single method) should be superimposed by certain behavior.

To our knowledge no approach, among those quoted, is interested in the adaptation of component structures. All are interested in the adaptation of services. This adaptation can be carried out in a static [18] or dynamic [19], [10] way. Binary component adaptation (BCA) [18] is a mechanism to modify existing components (such as Java class files) to the specific needs of a programmer. Binary component adaptation (BCA) allows components to be adapted and evolved in binary form and on-the-fly (i.e. during program loading). Concerning the second criterion related to restructuring approaches, we can quote refactoring techniques [21], [23] which

[3]Tool used to deliver identifiers which aims at ordering events or transactions.

the aim is to restructure an existing body of an object-oriented code, altering its internal structure without changing its external behavior [21]. Generally, refactoring is used to make the code simpler in order to include or understand it more easily [13]. It also allows to find the potential bugs or errors more quickly. It makes it possible to eliminate the duplicated code. The goal of this technique is to reorganize classes, variables and methods in a new hierarchy in order to facilitate its future adaptation or extension [12]. Its use increases the program quality (e.g. reusability, effectiveness, updating, etc.).
Another technique of program analysis is slicing [1]. It is generally used for the code debugging and testing [2], for maintaining [14] or for transforming source code. The goal of this technique is to determine the programs behavior but also that of all elements which it can contain (e.g. variables, methods, etc.). For example, slicing can allow to detect all instructions which can affect a variable. In fact, this technique is related to the dependence graph of a program [17].

## 5. CONCLUSION

We presented in this paper an approach to dual with the adaptation of software components. This approach is based on the consideration of a new adaptation facet: the structure of a software component. This approach is implemented and a prototype has been developed using the Julia [7] software component framework. This last is a Java implementation of the Fractal component model [8]. We have developed, in addition to the structural adaptation process presented here, a component model supporting this process. This model was not presented here due to space limitation.
Our approach need source code analysis and instrumentation. It not consider run-time adaptation problem. However, it's enough generic to be applicable for dynamic adaptation. Nevertheless, concerning this possibility, it is necessary to define, in addition to the already defined process, mechanisms for the management of the dynamicity (e.g. disconnection, connection, interception of the invocations of services, service recovery, etc). Thus, the dynamicity management constitutes one direction of our future work.
As we noted it before, many applications of the structural adaptation approach are possible. We expect, for a better management of system resources availability, developing mechanisms permitting to use structural adaptation approach for the mobile and ubiquitous applications [20].

## 6. REFERENCES


[1] H. Agrawal and J. Horgan: Dynamic program slicing. In Proc. of the ACM SIGPLAN '90 Conference, 1990.

[2] H. Agrawal, R. Demillo, and E. Spafford: Debugging with dynamic slicing and backtracking. Software-Practice an Experience, 23(6): 589-616, 1993.

[3] D. Ayed, C. Taconet and G. Bernard: Component-oriented approaches to context-aware systems. Workshop ECOOP'04, Oslo, Norway, 2004.

[4] D. Balek: Connectors in Software Architectures. Ph.D. Thesis, advisor: Frantisek Plasil, 2002.

[5] J. Bosch: Superimposition: A Component Adaptation Technique. Information and Software Technology, 1999.

[6] A. Brogi, C. Canal, E. Pimentel: Behavioural types and component adaptation. 10th International Conference on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science 3116, 42-56, 2004.

[7] E. Bruneton: Julia Tutorial. http://fractal.objectweb.org/tutorials/julia/

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, J.-B. Stefani: An Open Component Model and Its Support in Java. CBSE, 7-22, 2004.

[9] L. Courtrai, F. Guidec, Y. Maho: Middleware Support for the Deployment of Resource-Aware Parallel Java Components on Heterogeneous Distributed Platforms, in 30th Euromicro Conference - Component-Based Software Engineering Track, Rennes, France, 2004.

[10] P.-C. David and T. Ledoux: Dynamic Adaptability of Services in Enterprise JavaBeans Architecture. ECOOP'02 Workshop on "Component-Oriented Programming", Malaga, Spain, 2002.

[11] H.-E. Eriksson: UML 2 Toolkit, Wiley edition, ISBN: 0471463612, 2003.

[12] B. Foote and W. F. Opdyke: Life Cycle and Refactoring Patterns that Support Evolution and Reuse. First Conference on Pattern Languages of Programs (PLOP '94), Monticello, Illinois, 1994. In Pattern Languages of Programs, Addison-Wesley, 1995.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: Refactoring: Improving the Design of Existing Code. ISBN 0201485672. (The Addison-Wesley Object Technology Series) Hit the shelves in mid-June of 1999.

[14] K. B. Gallagher and J. R. Lyle: Using program slicing in software maintenance. IEEE Transactions on Software Engineering, 17(8):751-761, 1991.

[15] Graham, Ian S.; Quin, Liam.: XML Specification Guide. New York, NY: John Wiley and Sons, 1999.

[16] G. T. Heineman and H. Ohlenbusch: An Evaluation of Component Adaptation Techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, 1999.

[17] S. Horwitz, T. Reps, and D. Binkley: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 26-60, 1990.

[18] R. Keller, U. Hlzle: Binary Component Adaptation. ECOOP, 307-329, 1998.

[19] A. Ketfi, N. Belkhatir, P.Y. Cunin: Automatic Adaptation of Component-based Software: Issues and Experiences. PDPTA'02, Las Vegas, Nevada, USA, 2002.

[20] N. Le Sommer, F. Guidec: A Contract-Based Approach of Resource-Constrained Software Deployment. Component Deployment, 15-30, 2002.

[21] T. Mens, T. Tourw: A Survey of Software Refactoring. IEEE Transactions on Software Engineering, V. 30, Number 2, 126-139, 2004.

[22] M. G. Nanda: Slicing Concurrent Java Programs: Issues and Solutions. Thesis. Indian Institute of Technology, Bombay, 2001.

[23] F. W. Opdyke: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992. Available as Technical Report No. UIUCDCS-R-92-1759.

[24] C. Szyperski: Component Software. Beyond Object-Oriented Programming - Addison-Wesley / ACM Press, 1998.