Transformation of Centralized Software Components into Distributed Ones by Code Refactoring

Abdelhak Seriai[1], Gautier Bastide[1], and Mourad Oussalah[2]

[1] Ecole de Mines de Douai, 941 rue Charles Bourseul, 59508 Douai, France
{seriai, bastide}@ensm-douai.fr

[2] LINA, universitẻ de Nantes, 2 rue de la Houssinière, 44322 Nantes, France
oussalah@lina.univ-nantes.fr

Abstract. Adapting software components to be used in a particular application is a crucial issue in software component based technology. In fact, software components can be used in contexts with characteristics different from those envisaged when designing the component. Centralized or distributed deployment infrastructure can be one of these assumptions. Thus, a component can be designed as a monolithic unit to be deployed on a centralized infrastructure, nevertheless the used infrastructure needs the component to be distributed. In this paper, we propose an approach allowing to transform a centralized software component into a distributed one. Our technique is based on refactoring and fragmentation of component source code.

Keywords: software component, adaptation, restructuration, distribution, refactoring.

1 Introduction

Component-based software engineering (CBSE) focuses on reducing application development costs by assembling reusable components like COTS (Commercial-Off-The-Shelf). However, in many cases, existing components can not be used in an ad-hoc way. In fact, using a software component in a different manner than for which it was designed is a challenge because the new use context may be inconsistent with assumptions made by the component. Deployment infrastructure may be one of these assumptions. For example, a software component may be designed as a monolithic unit to be deployed on a centralized infrastructure and, due to load balancing performance, security policy or other motivations, this component has to be distributed. The solution consists in adapting this component to its distributed use context.

Therefore, we propose in this paper, an approach aiming at transforming an object-oriented monolithic and centralized software component by integrating distribution facilities. Our approach is based on two transformations. The first one consists in refactoring component structure in order to create a composite-component (i.e. fragmented structure), while preserving component's behaviour. This transformation is achieved through a process composed of four stages. First, following the available infrastructure, the needed distribution configuration is expressed in a declarative style. Next, the monolithic component is fragmented to fulfil the distribution specification given during the first stage. After, components generated as fragmentation result are assembled. Finally, the component assembly is wrapped into a composite-component which is integrated into the application.

The second transformation makes the generated composite-component distributed. In fact, the refactoring process applied to a monolithic centralized component generates a composite one but still with centralized constituents. So, in order to create a distributed composite-component, we need to transform local composition links between its constituents into remote ones. Remote links reflect the distributed configuration specified for the adapted component services.

We discuss the proposed approach in the rest of this paper as follows. In section 2, we present an example of experimentation that illustrates our approach. Section 3 and 4 detail respectively, the refactoring process allowing to fragment a component and next the integration of the distribution mechanisms. Section 5 reviews related works. Conclusion and future works are provided in section 6.

2 Example of Illustration: A Shared-Diary Component

In order to illustrate our purpose, we use throughout this paper an example of a monolithic software component providing services of a shared-diary system accessible to multiple users. It allows to store and consult the personal diaries of each member of a group and it coordinates dependent events stored or generated by these diaries. The *shared-diary* component provides the following services:

1. Managing personal diary. This includes authentication, consulting events (e.g. meeting, activities, projects, etc.), 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, 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 diary 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.

We consider that this component is a monolithic and centralized one. Also, we assume that, due to the considered load balancing policy, defined for the available deployment infrastructure, this component cannot be deployed on only one host. So, our goal is to transform this component for deploying it on a distributed infrastructure (Fig. 1). This result may be obtained by the fragmentation of the *shared-diary* component into four new components called *diary-manager*, *database-manager*, *absence-manager* and *meeting-manager* which may be deployed on distinguished hosts.

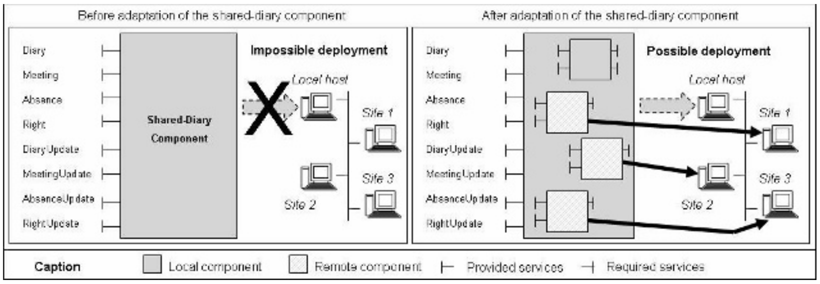


Fig. 1. Transformation of the *shared-diary* component into a distributed one

3 From a Monolithic Component to a Composite-Component

The first transformation to obtain a distributed component from a monolithic centralized one consists in refactoring component code through the fragmentation of its structure. As we have mentioned it previously, the component refactoring process (Fig. 2) is based on four stages which are detailed below.

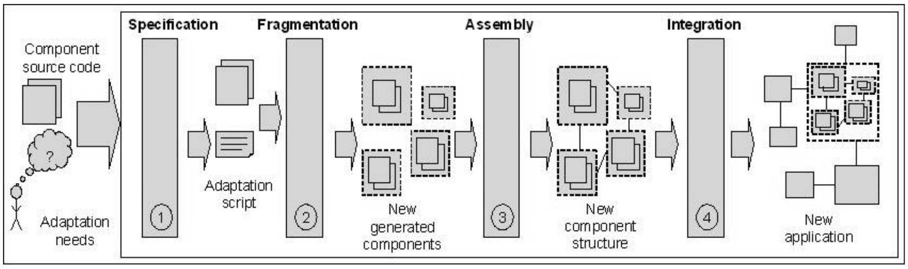


Fig. 2. Software component refactoring process

3.1 Specification of the Transformation Result

This first stage aims at indicating how services provided by the component to be transformed are to be deployed on the available distributed infrastructure. This is done by specifying for every provided service, its deployment host. This operation is realized using a script defining components to be generated and for each component, its provided interfaces. The script syntax[1] is given bellow.

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

To illustrate this, let us reconsider our example of the shared-diary application, the goal of this component transformation is to reorganize services provided by this one in four new generated components (e.g. *Diary-Manager*, *DataBase-Manager*, *Absence-Manager*, *Meeting-Manager*). The *Diary-Manager* component (provided interfaces: *Diary* and *DiaryUpdate*) will be deployed on the local site whereas the *DataBase-Manager* component (provided interfaces: *Right* and *RightUpdate*), *Absence-Manager* component (provided interfaces: *Absence* and *AbsenceUpdate*) and *Meeting-Manager* component (provided interfaces: *Meeting* and *MeetingUpdate*) are deployed respectively on *site1*, *site2* and *site3*. The script allowing to obtain the needed structure is the following:

```
 StructuAdapt (Shared-Diary,
 {Diary-Manager=<{P-Diary=Diary,DiaryUpdate}>}
 {DataBase-Manager=<{DB=Right, RightUpdate}>,<site1>}
 {Absence-Manager=<{P-Absence=Absence,AbsenceUpdate}>,<site2>}
 {Meeting-Manager=<{P-Meeting=Meeting,MeetingUpdate}>,<site3>}
)
```

3.2 Component Fragmentation

Specification done during the previous stage is used to refactor component structure. Component refactoring consists in fragmenting this component into a set of new generated components, while guaranteeing the component integrity and coherence. This stage is based on component code analysis.

Fragmentation Control: Component code refactoring must be realized without any change on this component's behaviour. Thus, two criteria must be checked: integrity of generated components and coherence of their respective states.

– *Generated component integrity.* The implementation of each component to be generated must be guaranteed to be sound. The soundness of this code[2]

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

[2] Proof of the satisfaction of these soundness criteria by the proposed refactoring approach is out of this paper scope.

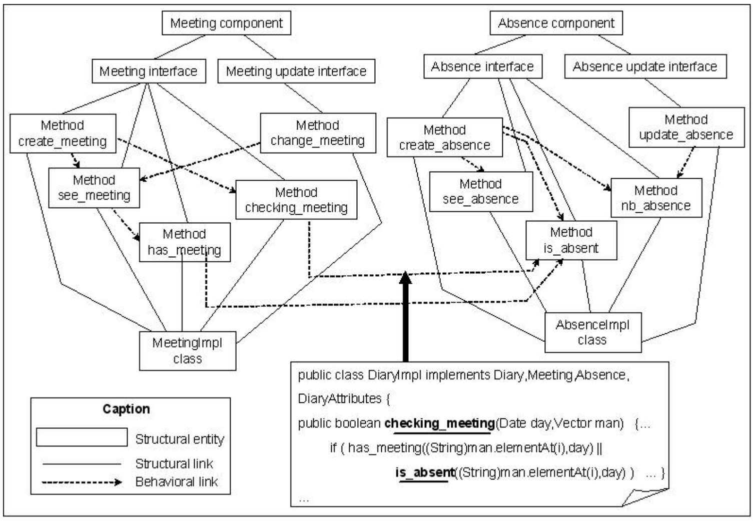
implies that it must be syntactically and semantically correct (i.e. code must be conform to the corresponding object-oriented grammatical and semantic language rules), complete (i.e. dependent code elements must be accessible one to the others) and coherent (i.e. the behaviour corresponding to a generated component must be conform to the matching local behaviour in the monolithic component).

– *Generated component coherence.* The outside behaviour made by the generated components must be the same as the monolithic component's behaviour. That implies that local behaviours of generated components must be coherent, the ones compared to the others. This requires that local behaviours corresponding respectively to the generated components which are semantically related to other behaviours in other components must be identified to ensure their correlation.

Code Analysis and Fragmentation: The fragmentation which aims at generating new software components is realized by analysing the monolithic component source-code, determining for each new component to be generated its corresponding code, separating these codes, one from the others, and determining existing dependencies between them. These steps are mainly based on building, for each component to generate, its SBDG (i.e. Structural and Behavioural Dependency Graph). A SBDG is a graph where nodes are structural elements and arcs are the different forms of dependencies existing between these elements. Structural elements may be external (e.g. ports, interfaces, implementation class and methods matched with services provided by these interfaces) or internal (e.g. internal methods and inner classes) ones. Dependencies between structural elements are of two types: structural and behavioural dependencies. Structural dependencies correspond to composition relationships between structural elements. Thus, a software component is structurally dependent of its ports; a port is structurally dependent of its interfaces, etc. Behavioural dependencies represent method calls defined in a method code. It should be noted that the polymorphism property related to an object-oriented code does not allow to identify, by a static analysis and in a deterministic way, all existing behavioural links between methods. Thus, we insert in a SBDG all possible behavioural links existing between these structural elements (i.e. methods).

Once, the SBDG corresponding to a component to be generated is built, the code of each one of its structural elements is generated. These codes are connected between them in order to reflect the existing structural links between their corresponding structural elements. All the generated code represents the first version of a new component source-code. The next version of the generated component source-code transforms behavioural links existing between methods defined respectively by two different SBDG on composition links between the corresponding components (see Sect. 3.3).

For example, figure 3 shows a part of the SBDG corresponding to the *Meeting-Manager* and *Absence-Manager* 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



```
public class DiaryImpl implements Diary,Meeting,Absence,
DiaryAttributes {
public boolean checking_meeting(Date day,Vector man)  {...
        if ( has_meeting((String)man.elementAt(i),day) ||
                    is_absent((String)man.elementAt(i),day) )   ... }
...
```

Fig. 3. A part of the *Shared-diary* component SBDG

in another interface, it is needed to create a behavioural link between the *Meeting* and *Absence* interfaces.

3.3 Assembly of the New Generated Components

The fragmentation stage generates unconnected components providing each one a sub-set from the initial component services. However, these services are not independents one from the others. In fact, they are linked through behavioural or resource sharing dependencies which are materialized through connections between generated components.

Connecting Components Via Behavioural-Dependency Interfaces: Components generated by fragmentation are connected using behavioural-dependency interfaces. These interfaces are used to materialize behavioural-dependencies between generated components according to the SBDG graph. Behavioural-dependency interfaces defined by a generated component are:

– Interfaces defining required behavioural-dependency services. These interfaces allow a component service to access all needed elements (i.e. methods) which are contained in other generated component implementations.
– Interfaces defining provided behavioural-dependency services. These services are those provided by this component and which are required by other components to assuring some of their services.

Connecting Components Via Resource-Sharing Dependency Interfaces: Components are also connected via interfaces used to manage resource sharing. We consider as resource every structural entity defined in the component code with an associate state. For example, instance and class attributes are considered as resources. Shared resources are those defined and used in two or more component implementations. So, we need to preserve a coherent state of these resources in all components sharing them (i.e. the same resource with the same state on all components). Coherence is ensured through two types of interfaces. The first one aims at permitting to communicate, between components, updates occurred on shared resources. The second interface type allows to guarantee a synchronized access to shared resources. The implementation of these communication interfaces is realized through the instrumentation of the object-oriented source-code corresponding to these services [2].

- Communication interfaces are:
  1. An interface defining required services permitting to notify shared-resource state updates. These services are defined as synchronous (i.e. every time when a shared resource is updated by a component, its execution can continue only after its state is updated by the other components sharing this resource). Component implementation is instrumented by adding notification code every time the shared resources updated.
  2. An interface defining provided services allowing to update shared resource states after this resource been updated by another component. Thus, component implementation is instrumented by adding code permitting to read new resource values and update the local resource copy.

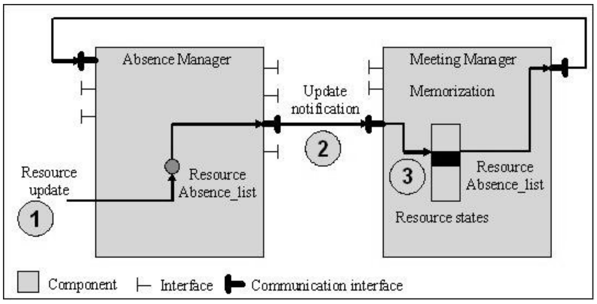


Fig. 4. Example of communication interfaces

Figure 4 shows an example of notification interfaces used to manage the *Absence_list* resource. This resource is an instance attribute whose value represents the absence days for a given person. It is shared by the *Absence-Manage* and *Meeting-Manager* components. When the *Absence_list* resource is updated by the *Absence-Manager* component (1), a notification is sent to the *Meeting-Manager* component (2). Then, this last one memorizes the new value (3).

- Synchronized access interfaces are:
  1. An interface defining required services permitting to acquire an authorisation to update shared resources, from components sharing these ones. These services are not called every time a shared resource is used in the component implementation code.
  2. An interface defining provided services allowing to release rights to update shared resources. These services are called by components sharing resources with the component providing this interface.

  Figure 5 shows an example of synchronized access interfaces used 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 components which share this resource (e.g. *DataBase-Manager* and *Diary-Manager*) (2). Then, after it receives a notification from these components, *Absence-Manager* can update the *nb_day_free* resource (3).

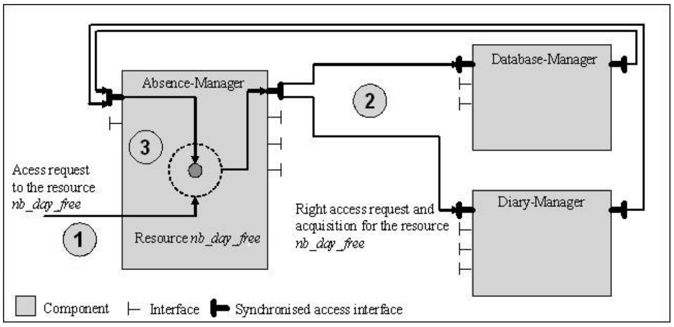

Absence-Manager
Database-Manager
Diary-Manager
2
3
1
Acess request to the resource nb_day_free
Resource nb_day_free
Right access request and acquisition for the resource nb_day_free
Component
Interface
Synchronised access interface

Fig. 5. Example of synchronized access interfaces

3.4 Integration of the Transformation Result

The last step of our process is the integration, in the subjacent application, of the component restructuring result obtained during the previous stages. It consists in connecting the new generated components with the other application components and to guarantee that the component transformation is achieved in a transparent way compared to the application components. In fact, the application must continue to be executed without any change compared to its initial configuration. So, integration requires to satisfy the following properties:

- Security condition: the application components should not be able to access, after the component transformation, to other services than those provided by the component before its transformation. In fact, all new interfaces (i.e. created by our process) must not be accessed by application components,

except those created by transformation. For example, all components must not access to services allowing to modify a shared resource state (i.e. only components which share this resource can access to related services).
- Distribution feature: New generated components can be accessed and handled as separate entities. For example, it would be possible to specify a deployment configuration by a direct designation of the generated components (i.e. components generated by fragmentation).

Our solution to guarantee these properties consists in encapsulating components generated by fragmentation into a new composite-component. This new component allows to mask access to "non functional" services (i.e. it wraps all the generated components). Moreover, it provides interfaces allowing to manipulate the generated components. For example, these interfaces aim at permitting independent deployment of each sub-component.

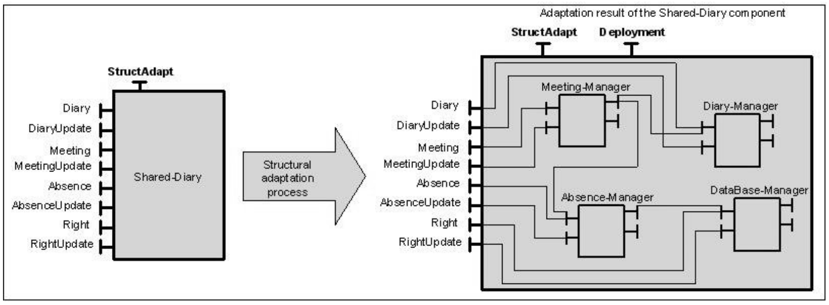


Fig. 6. Integration of component transformation result

4 From a Centralized Composite-Component to a Distributed Composite-Component

The fragmentation process realized during the first phase of our approach allows us to generate a new composite-component. However, this result cannot be distributed on several hosts because all sub-components use local binding. As many resources or services cannot be accessed using direct references because they are provided by remote components (i.e. sub-components are interconnected through bindings which can be local or remote references between provided and required interfaces), we need to ensure communications between local and remote components. In order to create distributed components, first, we need to specify the new component distribution (i.e. to specify sites for each component). This specification is realized through ADL generation (see Sect. 3.1). Then, the component structure is automatically updated (i.e. creation of new interfaces and components dedicated to the distribution management) and component code is instrumented in order to ensure coherence (i.e. a component may access to all resources or services needed during its execution).

In order to introduce distribution mechanisms into the composite-component generated during the first transformation process, we propose a distribution model for composite-components (Fig. 7). This model is composed of two parts. The first part is dedicated to the distribution management at the component content scale (i.e. new created interfaces and new added sub-components) and the other one defines all components needed at the controller scale (i.e. low-level services, network services, etc.).

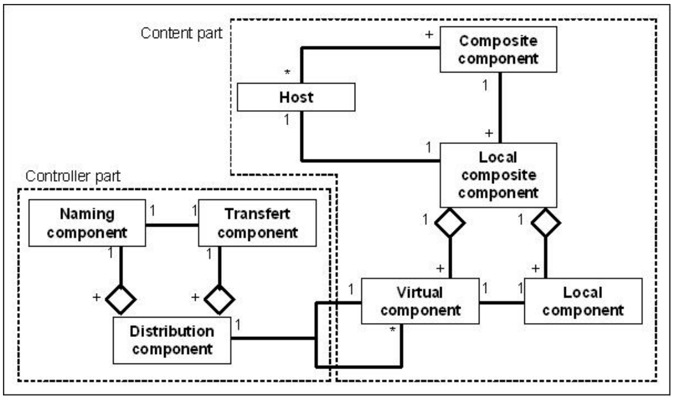


Fig. 7. Component distribution model

4.1 Distributed Composite-Component

A distributed component is a component whose sub-components may be deployed on different hosts. We distinguish three solutions which can be used to create a distributed component. The first one (see Fig. 8 Case B) consists in deploying sub-components on different hosts and the composite on only one. In this case, the composite-component instance contains only connectors which are used to transfer messages from provided composite ports (or interfaces) to sub-component ports (or interfaces) which may be provided by a local or a remote host (i.e. export binding). Moreover, sub-components are connected together through direct binding which may be local or remote ones. This strategy implies that sub-components may be accessible by a direct way. Moreover, the visibility of the internal composite structure is blurred. The second solution (see Fig. 8 Case C) consists in the use of virtual components within the composite. Virtual components are used in order to access a remote component (see below). This strategy allows to improve composite structure visibility. The last solution (see Fig. 8 Case A) consists in the creation of a composite-component into every host on which a part of the component is deployed. This solution allows to preserve a strong encapsulation of the created components. A composite-component instance is loaded on each host which contains a part of this component (i.e. at least one sub-component). Nevertheless, the entire composite-component is not instancied on each host. In fact, different copies of the composite-component are

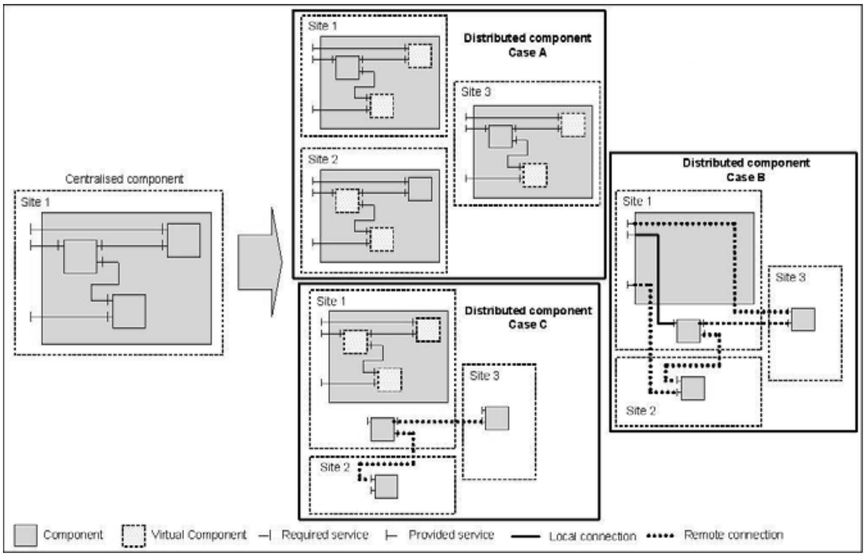


Fig. 8. Transformation from a centralized component to a distributed one

instancied. Each instance is composed of a set of local components and a set of virtual components.

Local Components: Local component means real component (i.e. sub-component) of the composite-component. They are generated during the fragmentation step of the first transformation. Each component is instancied in only one host (i.e. those which are specified by the administrator during the specification step).

Virtual Components

Virtual component structure: A virtual component provides the same interfaces than those of the remote component, however implementation (i.e. service code) is different. In fact, functional code is replaced by controller code which allows to invoke remote services. Two interfaces are added to this virtual component (Fig. 9): one is required and allows the component to send messages to the remote component and the other one is provided and allows the component to receive messages from the remote component. These two interfaces ensure remote communications. Bindings between virtual components are created using architecture description analysis (i.e. ADL analysis). For example, when a local component C1 deployed on site 1 is bound to a remote component C2 deployed on site 2 (i.e. a required interface of the component C1 is linked to a provided interface of the component C2), we create two links: one from the provided interface of the component C'2 (i.e. virtual component of C2 on site 1) to the required interface of the component C'1 (i.e. virtual component of C1 on site 2) and the other one from the provided interface of the component C'1 to the required interface of the component C'2. Communications between C'1 and C'2 components are realized through these two new interfaces whose services use the distribution components (see Sect. 4.2).

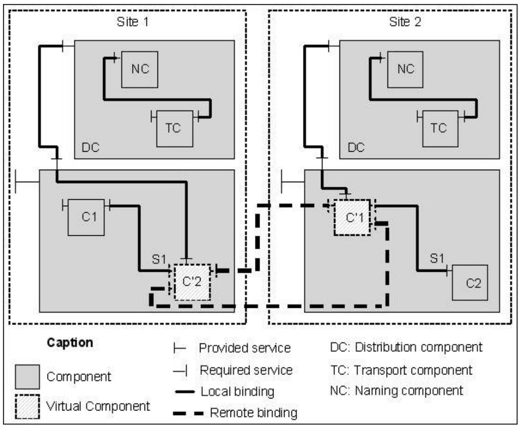


Fig. 9. Example of component distribution

Virtual component behaviour: A virtual component is a representation of a local component which is deployed on a remote host. In fact, it is used as connectors between local and remote components. Indeed, a local component service may invoke a remote service as if this one is provided by a local component (i.e. functional code of local components is not modified). Virtual components are used in order to transfer messages between local and remote components (i.e. delegation services). So, remote connections are realized only from a virtual component to another one because only these components are able to send and receive messages through network (Fig. 10). Thus, when a service of a component C1 calls a service provided by a remote component C2, the component C1 sends a message to the virtual component of C2. Then, this call is transformed into a call from the virtual component of C1 to the component C2. This transformation is realized through a remote connection between the virtual component of C2 and the virtual component of C1 (i.e. on the remote host).

4.2 Distribution Components

A new controller component called distribution component which allows to ensure remote communications is added to our model. It is composed of two subcomponents:

– A transport component: it allows virtual components to realize remote communications (i.e. services provided by the transport component allow to pack and unpack messages which are exchanged between local and remote components, and set up connections through network protocols).
– A naming component: it allows the transport component to find the host address on which local component services are instancied (i.e. services provided by the naming component allow to search and locate remote components).

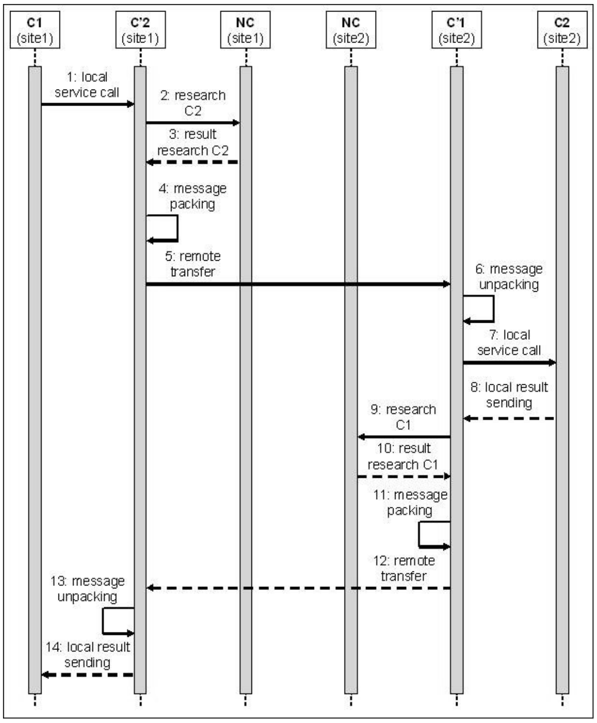


Fig. 10. UML2 Sequence Diagram of the distribution process between two components

As we explained previously, different component instances are loaded on deployment hosts. As a copy of the composite-component is created on each site, non-functional services (i.e. service allowing to manage component content, service allowing to manage bindings between components, service allowing to manage component life cycle, etc.) are duplicated. So, we need to ensure communication and coherence between component instances at the control scale in order to preserve software component integrity. For example, when the composite-component starts (i.e. call to the life cycle controller services), the other instances loaded on the remote hosts have to start their own component version. This operation can be realized using code instrumentation of controller services.

5 Related Works

We classify related works according to two criteria: the approach goal and the technique used to reach this goal. First, we present works related to software component adaptation. Next, we focus on works related to program transformation and restructuring and particularly those interested to object-oriented softwares.

Concerning the first criterion related to the adaptation goal, many adaptation approaches have been discussed in the literature [10]. Adaptation techniques can be categorized as either white-box or 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 [3] is an alternative technique. The idea behind is that the entire functionality of a component (i.e. rather than that of a single method) should be superimposed by certain behaviour.

To our knowledge no approach from those discussed in the literature, is interested in the adaptation of component structures. All are interested in service adaptation. This adaptation can be carried out in a static [11] or dynamic [12] way. Binary component adaptation (BCA) [11] is a mechanism to modify existing components (such as Java class files) to the specific needs of a programmer. It 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 [13] that aim at restructuring an existing body of an object-oriented code, altering its internal structure without changing its external behaviour. Generally, refactoring is used to make the code simpler in order to include or understand it easier [8]. It also allows to find the potential bugs or errors more quickly. It makes it possible to eliminate the duplicated code. This technique aims at reorganizing classes, variables and methods in a new hierarchy in order to facilitate its future adaptation or extension [7].

Another technique of program analysis is slicing [14]. It is generally used for the code debugging and testing [1], for maintaining [9] or for transforming source code. The goal of this technique is to determine program behaviour but also that of all elements which it can contain (e.g. variables, methods, etc.). For example, slicing allows to detect all instructions which can affect a variable.

6 Conclusion and Future Works

We presented in this article an approach allowing to create distributed components from monolithic ones. Our proposal is based on a new adaptation technique allowing to reorganize the software component structure using code refactoring. In fact, as we explained, component deployment and execution are linked to its structure. So, we propose to use this approach in order to fragment existent components and generate new components which can be distributed on several hosts. This approach is implemented and a prototype has been developed using the Julia [5] software component framework which is the Java implementation of the Fractal component model [4]. Fractal and Julia are developed by the INRIA[3]. Fractal is a hierarchical component model quite close to that proposed by UML2 [6].

Our approach needs source code analysis and instrumentation. It does not consider run-time adaptation problems. However, it is generic enough to be applied to dynamic adaptation. Nevertheless, concerning this possibility, it is necessary to define, in addition to the presented process, mechanisms for the dynamicity management (e.g. disconnection, connection, interception of the invocations of services, service recovery, etc). Thus, this way constitutes one direction of our future work.

As we noted it before, the main application of our approach consist in realizing a flexible deployment of software components. A future work may consist in the deployment process automation according to the execution context.

References

1. H. Agrawal, R. Demillo, and E. Spafford: Debugging with dynamic slicing and backtracking. Software-Practice an Experience, 23(6): 589-616, 1993.
2. G. Bastide, A-D. Seriai, M. Oussalah: Adapting Software Components by Structure Fragmentation. The 21st Annual ACM Symposium on Applied Computing; Software Engineering: Applications, Practices, and Tools (SE), Dijon, France, April 2006.
3. J. Bosch: Superimposition: A Component Adaptation Technique. Information and Software Technology, 1999.
4. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.-B. Stefani: An Open Component Model and Its Support in Java. CBSE, 7-22, 2004.
5. E. Bruneton: Julia Tutorial. http://fractal.objectweb.org/tutorials/julia/
6. H.-E. Eriksson: UML 2 Toolkit, Wiley edition, ISBN: 0471463612, 2003.
7. 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.
8. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: Refactoring: Improving the Design of Existing Code. ISBN 0201485672, 1999.
9. K. B. Gallagher and J. R. Lyle: Using program slicing in software maintenance. IEEE Transactions on Software Engineering, 17(8):751-761, 1991.
10. 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.
11. R. Keller, U. Holzle: Binary Component Adaptation. ECOOP, 307-329, 1998.
12. A. Ketfi, N. Belkhatir, P.Y. Cunin: Automatic Adaptation of Component-based Software: Issues and Experiences. PDPTA'02, Las Vegas, Nevada, USA, 2002.
13. T. Mens, T. Tourwe: A Survey of Software Refactoring, IEEE Transactions on Software Engineering, Volume 30, Number 2, pp. 126-139, February 2004.
14. M. Weiser: Program Slicing. IEEE Trans. Software Eng. 10(4): 352-357, 1984.

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