Automated and Unanticipated Flexible Component Substitution

Nicolas Desnos[1], Marianne Huchard[2], Christelle Urtado[1], Sylvain Vauttier[1], and Guy Tremblay[3]

[1] LGI2P - Ecole des Mines d'Alès, Nîmes, France
[2] LIRMM - UMR 5506 - CNRS and Univ. Montpellier 2, Montpellier, France
[3] Département informatique, UQAM, Montréal, QC, Canada

{Nicolas.Desnos, Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr, huchard@lirmm.fr, tremblay.guy@uqam.ca

Abstract. In this paper, we present an automatic and flexible approach for software component substitution. When a component is removed from an assembly, most existing approaches perform component-to-component substitution, relying on the fact that such a candidate component is available, which is hardly to happen because the constraints on its interfaces are too strong. When such a component does not exist, it would be more flexible to allow a single component to be replaced by a whole component assembly. We propose such an automatic substitution mechanism which does not need the changes to be anticipated and preserves the quality of the assembly.

1 Introduction

Today, software systems are becoming voluminous and complex like never before. Component-based software engineering [1] is a good solution to optimize the time and cost of software design while still guaranteeing the quality of the software. Moreover, the modularity it enables allows to tame the complexity of large systems. Typically, a component is seen as a black box which provides and requires services through its interfaces. An architecture is built to fulfill a set of functional objectives (its functional requirements)[4] and is described as a static interconnection of software component classes. A component assembly is a runtime instantiation of an architecture composed of linked component instances.

In this paper, we present an automatic and flexible approach for dynamic software component substitution. Anticipating component substitution, to overcome component obsolescence, failure or unavailability, is not always (cognitively) possible. Repairing a component assembly after a component has been removed while preserving its whole set of functionalities is difficult. When a component is removed from an assembly, most existing approaches perform component-to-component substitution [2–5]. However, these approaches rely on the fact that such an appropriate component, candidate for substitution, is available. This

[4] Our work does not yet handle non-functional requirements.

situation is hardly to happen because it is difficult to find a component that has the same capabilities than the removed one.

When such a component does not exist, allowing a single component to be replaced by a whole component assembly would permit more flexibility. In this paper, we propose such an automatic substitution mechanism which does not need the changes to be anticipated. Our approach relies on primitive and composite ports for replacing a component by a whole assembly of components while preserving the quality of the assembly.

The rest of this paper proceeds as follows. Section 2 introduces component-based software engineering, presents existing work on component substitution and shows their limits. Section 3 describes our proposition for dynamically replacing a component. We first shortly present how ports allow us to automatically build valid assemblies [6]. We then show how this process can be used as part of a flexible component substitution process. We also present how it is possible to simplify the assembly by removing all the components that have become useless. Finally, Section 4 concludes and proposes perspectives to this work.

2 Context and Related Work

2.1 Software Architecture Correctness and Completeness in CBSE

Component-Based Software Engineering [7] makes it possible to build large systems by assembling reusable components. The life cycle of a component-based architecture can be divided into three phases: design-time, deployment-time and runtime.

At *design-time*, the system is analyzed, designed and the validity of the design is checked. An architecture is built to fulfill a set of functional objectives (its functional requirements) [8, 9]. Functional objectives are defined as a set of functionalities to be executed on selected components. Selecting the functional objectives is typically a task performed during the analysis step. The structure of the architecture is described, during the design step, as a static interconnection of software component classes through their interfaces. It requires both selecting and connecting[5] the software components to be reused. This description, typically written in an architecture description language [10], expresses both the functional and non-functional capabilities of the architecture, as well as both the structural and the behavioral dependencies between components. Once the architecture is described, its validity is statically checked. Most systems verify the correctness of the architecture; some also guarantee its completeness.

Correctness. Verifying the correctness of an architecture amounts to verifying the connections between components and checking whether they correspond to a possible collaboration [9]. These verifications use various kinds of meta-information (types, protocols, assertions, etc.) associated with various structures

[5] We assume that the selected components need no adaptation (or have already been adapted).

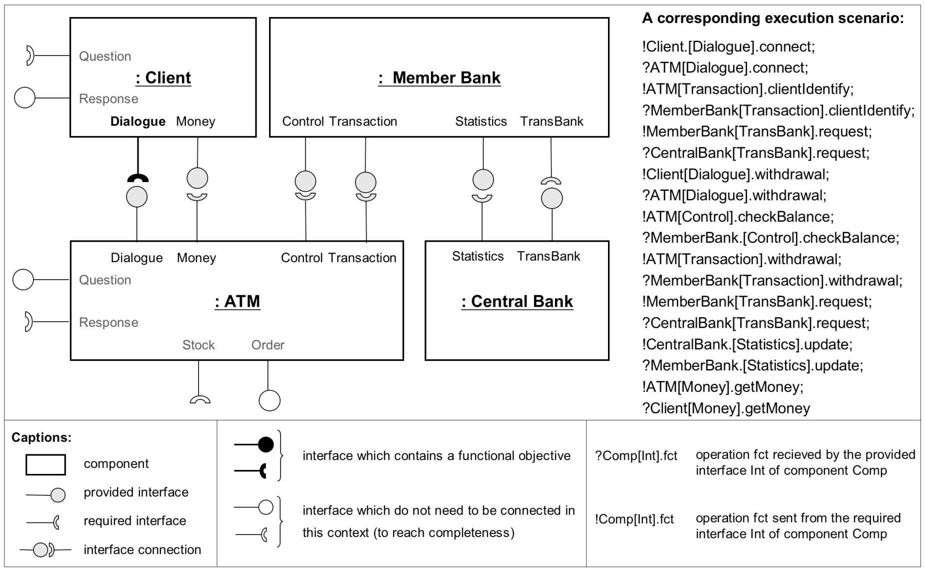
(interfaces, contracts, ports, etc.). The finest checks are done by protocol comparisons, which is a combinatorial problem [11–13].

Completeness. The architecture must also guarantee that all its functional objectives are going to be supported. In other words, the connections of an architecture must be sufficient to allow the execution of collaborations that reach (include) all the functional objectives. We call this **completeness** of the architecture [6]. Indeed, the use of a component functionality (modeled by the connection of an interface) can necessitate the use of other functionalities which, in turn, entail new interface connections. Such functionalities (or interfaces) are said to be **dependent**. This information is captured in the description of component behavior and depends on the context in which the functionality is called (execution scenario). There are various ways to ensure completeness:

- For a first class of systems [14], completeness of an architecture is guaranteed by verifying that all the interfaces of all its components are connected. This view makes checking completeness very simple but over-constrains the assembly thus diminishing both the capability of individual components to be reused in various contexts and the possibilities of building a complete architecture, given a set of predefined components.
- To overcome the defects of the first class of systems, a second class of systems [3] defines two categories of interfaces (mandatory and optional). These systems allow complete architectures to be built while still leaving pending interfaces (the optional ones). This view does not complicate the checking of completeness and increases the opportunities of building a complete architecture, given a set of predefined components. However, associating the mandatory / optional property to an interface regardless of the assembly context does not increase the capability of individual components to be reused in various contexts.
- The third strategy requires connecting only the interfaces which are strictly necessary to reach completeness [12, 15, 16] by exploiting the description of the component behavior. This is typically done by analyzing protocols which makes completeness checking less immediate.

Example. Figure 1 illustrates that it is possible to ensure completeness of an assembly while connecting only the strictly necessary interfaces. The *Dialogue* interface from the *Client* component represents a functional objective and must therefore be connected. As deducted by analyzing the execution scenario that has to be supported, all the dependent interfaces (grayed on Figure 1) must also be connected in order to reach completeness. For example, the *Control* interface from the *MemberBank* component must be connected whereas the *Question* interface from the *Client* component does not need to be connected.

Once the validity of the architecture is checked, it can be deployed (*deployment-time*). Deployment requires instantiating the architecture, configuring its physical execution context and dispatching the components in this physical context. One of the results of deployment is a component assembly: a set of linked component instances that conforms to the architectural description.



A corresponding execution scenario:

```
!Client.[Dialogue].connect;
?ATM[Dialogue].connect;
!ATM[Transaction].clientIdentify;
?MemberBank[Transaction].clientIdentify;
!MemberBank[TransBank].request;
?CentralBank[TransBank].request;
!Client[Dialogue].withdrawal;
?ATM[Dialogue].withdrawal;
!ATM[Control].checkBalance;
?MemberBank.[Control].checkBalance;
!ATM[Transaction].withdrawal;
?MemberBank[Transaction].withdrawal;
!MemberBank[TransBank].request;
?CentralBank[TransBank].request;
!CentralBank.[Statistics].update;
?MemberBank.[Statistics].update;
!ATM[Money].getMoney;
?Client[Money].getMoney
```

Fig. 1. A complete assembly and a possible corresponding execution scenario

At *runtime*, the component assembly executes. The evolution of this assembly is an important issue for the application to adapt to its environment in such situations, as maintenance, evolution of the requirements, fault-tolerance, component unavailability in mobile applications, etc. In this context, an important question is: What are the possible dynamic evolutions that can be supported by the component assembly and by the architecture itself? The remaining of this paper is a tentative answer to this question.

2.2 Dynamic Architecture Reconfiguration

To ensure that a component assembly will remain valid at runtime, all systems try to control how the assembly evolves. Different evolution policies exist:

– The simplest and most restrictive is to forbid dynamic reconfigurations: assemblies cannot evolve at runtime. This policy is not satisfactory.
– Some systems [17, 3] allow the structure defined in the architecture to be violated when modifying component assemblies at runtime. They authorize component and connection modifications (addition, suppression) based on local interface type comparisons. The result is a lack of control on the assembly: its validity is not guaranteed anymore.
– The third category of systems ensures that component assemblies always conform to the structure defined in the architecture. All the possible evolutions must therefore be anticipated at design-time and described in the architecture itself [10]. Different techniques are used. ArchJava [18] and Sofa 2.0 [5] use patterns to know which interfaces can be connected or disconnected and

which components can be added or removed. Others [19, 20] use logical rules that are a more powerful means to describe the possible evolutions. These solutions complicate the design process and make anticipation necessary while it is not always (cognitively) possible [5, 21].

Dynamic Component Removal. Among the situations to handle to enable component assembly evolution is dynamic component removal. When removing a component from an assembly, the main issue is to ensure that there will not be any functional regression. The third category of systems typically allow a removed component to be replaced by a component which provides compatible services in order for the asssembly to still conform to the architecture. The anticipation of the possible evolutions allow these systems to ensure that the new component assembly will still satisfy the validity property that has been checked statically on the architecture at design-time. There are two major interpretations of component compatibility. In most of the systems [22, 2, 5, 3], the components must strictly be compatible: the new component must provide at least all the provided interfaces the removed component did and it cannot require more required interfaces. In [23], compatibility is less restrictive and context-dependent. If a provided interface from the removed component is not used by another component in the assembly (not used in this context), the new component is not required to provide this interface (as it is not necesssary in this context). On the other hand, the new component can have extra required interfaces as soon as those interfaces find a compatible provided interface among the components of the assembly. This context-dependent definition of component compatibility allows better adaptability of the component assemblies.

Discussion. There are two main restrictions to the state of the art solutions to complete a component assembly after a component has been dynamically removed:

1. Anticipating all possible evolutions to include their description in the initial description of the architecture at design-time is not always possible because it requires to know all the situations that may occur in the future of the system. Ideally, it should be better to try and manage the evolution of software assemblies in an unanticipated way.
2. Replacing the removed software component by a single component is not always possible because it is quite unlikely that a component having compatible interfaces exist among the potential candidates for substitution. In the more general case when such an adequate component does not exist, it might be interesting to replace the removed component by a set of linked components that together can provide the required services.

Proposing a solution to replace a removed component by an assembly of components in an unanticipated way while trying as much as possible to guarantee the quality (executability) of the assembly is the initial motivation for the work presented in this paper.

3 Automated and Unanticipated Flexible Component Substitution

In previous work, we proposed [24] and optimized [6] a solution to automatically build component assemblies from components, given a set of functional objectives. The building process uses ports, which are extra information we suggest to add to components, and guarantees that the suggested assemblies are complete.

The idea we develop in this paper is to use this building process in order to re-build an assembly after a component has been removed, thus replacing a single component by a whole sub-assembly which is a more flexible solution. This can be done in four steps: (1) removing the target component, (2) removing all the (consequently) dead components, (3) re-build the assembly by adding new components and new bindings until the assembly is complete and (4) checking the validity of the suggested assembly.

In the remaining of this section, we first briefly present how primitive and composite ports are abstract concepts that embody the information needed to automatically build complete assemblies and describe the automatic building process. We then try to formalize the building process and rely on this formalization to describe how it can be used for component substitution (steps 2 and 3 listed above).

3.1 Building Valid Component Assemblies from Port Enhanced Components

This section briefly describes how adding ports to components provides a means to automatically build complete component assemblies [24, 6]. Existing approaches usually statically describe architectures in a top-down manner. Once the architecture is defined, they verify its validity using costly validity checking algorithms [11–13]. Our building of assemblies from components obeys an iterative (bottom-up) process. This makes the combinatorial cost of these algorithms critical and prevents us from using them repeatedly, as a naive approach would have suggested to. To reduce the complexity, we chose to simplify the information contained in protocols and to represent it in a more abstract and usable manner through primitive and composite ports. Ports allow us to build a set of interesting complete assemblies from which it is possible to choose and check the ones that are best adapted to the architect's needs.

Primitive and Composite Ports. The idea for building a complete component assembly is to start from the functional objectives and to select the suitable components and make necessary links between them. Completeness is a global property that we are going to guarantee locally, in an incremental way all along the building process. The local issue is to determine which interfaces to connect and where (to which component) to connect them. This information is hidden into behavior protocols where it is difficult to exploit in an incremental assembly process. We are going to enhance the component model with the notion of port, in order to model the information that is strictly necessary to guarantee completeness in an abstract way. Primitive and composite ports will therefore represent two kinds of connection constraints on interfaces, so that the necessary connections can be correctly determined. In some way, ports express the different usage contexts of a component, making it possible to connect only the interfaces which are useful for completeness. As in UML 2.0 [25], one can also consider that the functional objectives of an architecture are represented by use cases, that collaborations concretely realize use cases and contain several entities that each play a precise role in the collaboration. Primitive and composite ports can be considered as the part of the component that enables the component to play a precise role to realize a given use case.

Primitive ports are composed of interfaces, as in many other component models [25, 26]. Ports are introduced as a kind of structural meta-information, complementary to interfaces, that group together the interfaces of a component corresponding to a given usage context. More precisely, a primitive port can be considered as the expression of a constraint to connect a set of interfaces both at the same time and to a unique component.

In Figure 2, the *Money_Dialogue* primitive port gathers the *Dialogue* and the *Money* interfaces from the *Client* component. It expresses a particular usage context for this component. The connection between two primitive ports is an atomic operation that connects their interfaces: two primitive ports are connected together when all the interfaces of the first port are connected to interfaces of the second port (and reciprocally). Thus, port connections make the building process more abstract (port-to-port connections) and more efficient (no useless connections). In this example, the *Money_Dialogue* primitive port from the *Client* component is connected to the *Money_Dialogue* primitive port from the *ATM* component.

Composite ports are composed of other ports. A composite port expresses a constraint to connect a set of interfaces at the same time but possibly to different components. In Figure 2, the *ATM* component has a composite port which is composed of the two *Money_Dialogue* and *Money_Transaction* primitive ports.

Like a designer has to do with protocols, ports have to be manually added to document the design of components; however, we are currently working on their automatic generation from behavior protocols.

Completeness of an Assembly as Local Coherence of its Components. Calculating the completeness of an already built component assembly is of no interest in an incremental building approach. Our idea is to better consider a local property of components. We call this property **coherence** and have shown [24] that it is a necessary condition for validity. Intuitively, we can see that when all components of an assembly are coherent, the assembly is complete. A component is said to be coherent if all its composite ports are and a composite port is coherent if its primitive ports are connected in a coherent way (see below).

More formally, completeness can be described after setting some preliminary definitions.

– An **interface** is characterized by *a set of operations*.

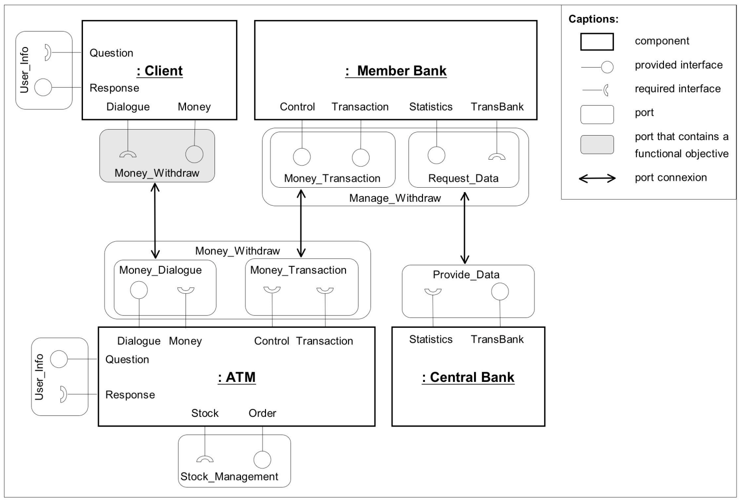


Fig. 2. Example of components with primitive and composite ports

– We define a **component** C as a quadruple:

$$C = (Prv_C, Req_C, Prim_C, Comp_C)$$

Prv_C is the set of C's provided interfaces and Req_C its set of required interfaces. $Prim_C$ is the set of C's primitives ports and $Comp_C$ its set of composite ports.

– We denote by $Int_C = Prv_C \cup Req_C$ the whole set of C's interfaces, and $Ports_C = Prim_C \cup Comp_C$ the whole set of C's ports.

– A **primitive port** ρ is a set of interfaces. For any primitive port ρ of C, $\rho \subseteq Int_C$. We denote by $\widehat{\rho}$ the fact that, with respect to a set of components, ρ is *connected*—i.e., any required (resp. provided) interface of ρ is correctly linked with a provided (resp. required) interface of another (primitive) port.

– A **composite port** γ of C is a set of ports, primitives or composites, from C.

– Let $\gamma \in Comp_C$ be a composite port of C. We define $PrimPorts^*(\gamma)$, resp. $CompPorts^*(\gamma)$, as the set of primitive, resp. composite, ports that are directly or indirectly contained in γ:

$$PrimPorts^*(\gamma) = \{\rho \in \gamma \cap Prim_C\} \cup \bigcup_{\gamma' \in \gamma \cap Comp_C} PrimPorts^*(\gamma')$$

$$CompPorts^*(\gamma) = \{\gamma' \in \gamma \cap Comp_C\} \cup \bigcup_{\gamma' \in \gamma \cap Comp_C} CompPorts^*(\gamma')$$

– We denote $\widehat{\gamma}$ when all primitive ports contained in γ are connected[6]:

$$\widehat{\gamma} = \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \cdot \widehat{\rho}$$

– We define a relation *Unrelated* between two different composite ports γ and γ' of Comp_C, denoting that neither port is directly or indirectly composed of the other:

$$\mathit{Unrelated}(\gamma, \gamma') = \gamma \neq \gamma' \wedge \gamma \notin \mathit{CompPorts}^*(\gamma') \wedge \gamma' \notin \mathit{CompPorts}^*(\gamma)$$

– Let $\gamma \in \mathit{Comp}_C$ be a composite port. $\mathit{Shared}(\gamma)$ is the set of primitive ports shared by γ and by another unrelated composite port of C:

$$\begin{aligned}\mathit{Shared}(\gamma) = \{\rho \in \mathit{PrimPorts}^*(\gamma) \mid \\ \exists\, \gamma' \in \mathit{Comp}_C \cdot \mathit{Unrelated}(\gamma', \gamma) \wedge \rho \in \mathit{PrimPorts}^*(\gamma')\}\end{aligned}$$

To determine the completeness of an assembly, we need to know if the interfaces that must be connected are indeed connected. The main idea is to check the coherence of each composite port. Two cases must be checked: when the composite port does not share any primitive ports with another unrelated composite port and when it does share some primitive ports.

Let us now define the coherence of a composite port. Given a composite port γ, three mutually exclusive cases are possible for γ to be coherent:

1. All its primitive ports are connected.
2. None of its primitive ports is connected.
3. Some, but not all, of its primitive ports are connected. In this case, γ can still be coherent if it shares some port with another unrelated composite port (of the same component) which is itself entirely connected. Indeed, sharing of primitive ports represents alternative connection possibilities [6]. A partially connected composite port can represent a role which is useless for the assembly as soon as its shared primitive ports are connected in the context of another (significant) composite port.

– Port γ is **coherent** if the following holds, where ρ is restricted to primitive ports of γ:

$$\oplus \begin{cases} \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \cdot \widehat{\rho} \qquad \text{(which is equivalent to } \widehat{\gamma}\text{)} \\ \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \cdot \neg\widehat{\rho} \\ \wedge \begin{cases} \forall\, \rho \in \mathit{Shared}(\gamma) \cdot \\ \quad \widehat{\rho} \Rightarrow \exists\, \gamma' \in \mathit{Comp}_C \cdot \mathit{Unrelated}(\gamma, \gamma') \wedge \rho \in \mathit{PrimPorts}^*(\gamma') \wedge \widehat{\gamma'} \\ \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \setminus \mathit{Shared}(\gamma) \cdot \neg\widehat{\rho} \end{cases} \end{cases}$$

– A component C is said to be **coherent** if all its composite ports are coherent:

$$\forall\, \gamma \in \mathit{Comp}_C \cdot \gamma \text{ is coherent}$$

– An assembly of components is said to be **complete** if *i*) all the primitive ports which represent functional objectives are connected; *ii*) all its components are coherent.

[6] As in VDM [27] and B [28], "·" is used to separate the (typed) variable introduced by the quantifier and the associated predicate.

Building Complete Component Assemblies. This coherence property allows us to concentrate on a local property of composite ports which is easier to include in an iterative assembly process. The principle of the automatic assembly process (detailed in [6]) is to try and connect all the ports representing a functional objective and iteratively discover and try to fulfill new connection needs. This process has been implemented as the searching of a construction tree using a depth-first policy. Backtracking is used to explore all the alternate construction paths (alternative possible components or alternative connection choices due to composite port intersections). This complete exploration of the construction tree is used to guarantee that any possible solution is always found. Furthermore, optimization strategies and heuristics have been added for the traversal of the construction space. The use of ports, and particularly of composite ports, is prominent in our approach: as they express the local dependencies that exist between interfaces, ports provide a simple means to evaluate the completeness of an architecture. As a result, the building algorithm provides a set of interesting complete architectures. Since architecture completeness is a necessary condition for architecture validity, the resulting set of complete architectures thus provides a reduced search space on which classical correctness checkers such as [5] are finally used on few selected assemblies.

3.2 Flexible Component Substitution using the Automatic Building Process

To react to the dynamic removal of a software component, we propose a two step process that allows a flexible replacement of the missing component:

1. analyze the assembly from which the component has been removed and remove the now useless (dead) components,
2. consider the incomplete component assembly as an intermediate result of our iterative building algorithm and therefore run the building algorithm on this incomplete assembly to re-build a complete assembly.

Removing the Dead Components. When a component has been removed from a complete assembly, there are parts of the assembly that become useless. Indeed, some of the components and connections in the original assembly might have been there to fulfill needs of the removed component. To determine which parts of the assembly have become useless, let us define a graph which provides an abstract view of the assembly.

An assembly can be represented as a graph where each node represents a component and each edge represents a connection between two (primitive) ports of two of its components. We also distinguish two kinds of components: those which fulfill a functional objective—i.e., the components which contain a port which contains an interface which contains a functional objective—and those which do not (cf. Figure 3).

An **assembly** A can then be seen as a graph along with a set of functional objectives:

$$A = (G_A, FO_A)$$

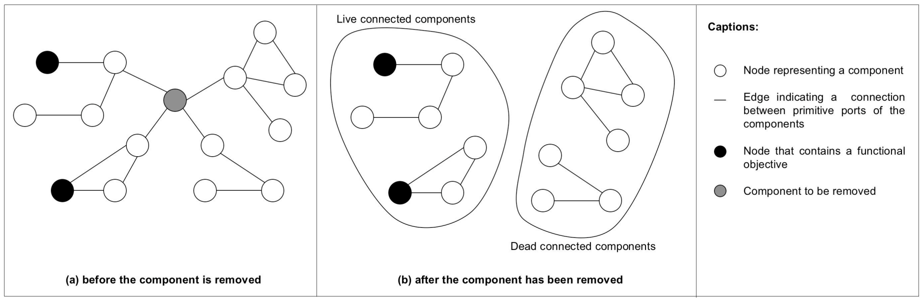

Live connected components
Dead connected components
Captions:
Node representing a component
Edge indicating a connection between primitive ports of the components
Node that contains a functional objective
Component to be removed
(a) before the component is removed
(b) after the component has been removed

Fig. 3. An assembly can be seen as an abstract graph (a) and divided in two sets of connected components when a component has been removed (b)

Here, $G_A = (Cmps_A, Conns_A)$ is a graph, with $Cmps_A$ the set of nodes—each node being a component, $Conns_A$ the set of edges—each edge indicating the existence of some primitive port connection between the components, and $FO_A \subseteq \bigcup_{C \in Cmps_A} Prim_C$ the set of primitive ports that contain some functional objectives[7].

If we consider the graph that results from the removal of the node representing the removed component, it is possible to partition it in two parts: the connected components[8] that have at least a node which contains a functional objective and the connected components that have no node that contains a functional objective. The second part of the graph is no longer useful because the components of this part of the graph were not in the assembly to fulfill functional objectives but to fulfill the needs of the removed component. Removing this part of the graph amounts to removing now useless parts of the assembly before trying to re-build the missing part with new components and connections.

Let $A = (G_A, FO_A)$ be an assembly and let $C \in Cmps_A$ be the component to remove. We define $H_{A,C}$ as the graph G_A from which we removed component C and all the edges (denoted by $Conns_C$) corresponding to primitive port connections between C and another component of G_A:

$$H_{A,C} = (Cmps_A \setminus \{C\}, Conns_A \setminus Conns_C)$$

We define $\mathcal{L}_{A,C}$ the live connected components of $H_{A,C}$ as the graph composed of all the connected components of $H_{A,C}$ that have at least a node which contains a functional objective.

We also define $\mathcal{D}_{A,C}$ the dead connected components of $H_{A,C}$ as the graph composed of all the connected components of $H_{A,C}$ that have no node which contains a functional objective.

[7] Recall that a functional objective is simply an operation defined in one of the provided interfaces.

[8] In this subsection of the paper, connected component refers to a subgraph that is connected, meaning that there exists a path between any of its two nodes.

Let us just notice that:

$$H_{A,C} = \mathcal{L}_{A,C} \cup \mathcal{D}_{A,C}$$

Figure 3 illustrates the definitions of $\mathcal{L}_{A,C}$ and $\mathcal{D}_{A,C}$. When a component is removed from the assembly, it is possible to remove all the components which do not participate any more to the completeness. Components from the dead connected components $\mathcal{D}_{A,C}$ can be removed from the assembly because they only participated to the coherence of the removed component.

Removing the dead components is a necessary step because keeping useless components add useless dependencies that make the resulting assembly considerably bigger thus complicating the building process, making the validity checks more difficult and making the assembly more subject to failures, less open for extensions, etc. Let us just also note that the components in $\mathcal{D}_{A,C}$ are dead components but that there still might be useless components in $\mathcal{L}_{A,C}$ (those we keep). We are thinking of future improvements on the detection of dead components that would better exploit the protocols.

3.3 Re-building the Removed Part from the Architecture

Once the dead components have been removed from the component assembly, the assembly contains all the components necessary to ensure completeness but one (the removed component) and its dependent components. Some of the dependencies of the remaining components are not yet satisfied. The issue is to find a component (like other systems do) or a series of assembled components that can fulfill the unsatisfied dependencies as the removed component did. We assume that it is quite unlikely that there exists a component that exactly matches the role the removed component had in the assembly. It is more likely (more flexible) that we have the possibility of replacing the removed component by a set of assembled components that, together, can replace the removed component.

In order to do so, we use the automatic building process presented in Section 3.1. The partial assembly in $\mathcal{L}_{A,C}$ is the starting point. It is considered as an intermediate result of the global building process. It is not complete yet: there still exist unsatisfied dependencies that were fulfilled by the removed component. The building process we described above is used to complete the architecture.

Evolution Scenario. On our *ATM* example, Figure 4 (a) represents the graph corresponding to the example of Figure 2. The *Client* node represents the *Client* component which contains a functional objective. The other nodes (*MemberBank*, *ATM* and *CentralBank*) represent components which do not contain any functional objective. Figure 4 (b) shows that the partial component assembly from $\mathcal{L}_{ATMexample,MemberBank}$ is not complete because the *ATM* component has become incoherent after the *MemberBank* component and the consequently dead components ($\mathcal{D}_{ATMexample,MemberBank} = \{CentralBank\}$) have been removed. To complete the assembly, new components must be added. Figure 4 (c) illustrates the result of this re-building process: The *IndependentBank* component

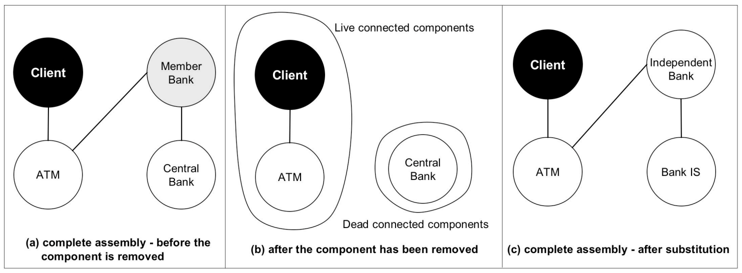

Client
Member Bank
ATM
Central Bank
(a) complete assembly - before the component is removed
Live connected components
Client
ATM
Central Bank
Dead connected components
(b) after the component has been removed
Client
Independent Bank
ATM
Bank IS
(c) complete assembly - after substitution

Fig. 4. Evolution scenario on the ATM example

is connected to the *BankIS* component and they both replace the components that had been removed to complete the *ATM* example assembly.

Figure 5 details the resulting architecture. In this example, the component to remove is the *MemberBank* component. When the *MemberBank* component is removed, completeness of the architecture is lost. Indeed, the *ATM* component is not locally coherent any more. Its *Money_Withdraw* composite port is not coherent because the primitive port *Money_Transaction* is not connected and the *Money_Dialogue* primitive port is connected. The *CentralBank* component constitutes the $\mathcal{D}_{ATMexample,MemberBank}$ graph and can also be removed. Completeness is researched by selecting and connecting new components. In this example, an *IndependentBank* component is connected to the *ATM* component through its *Money_Transaction* primitive port. At this step, the assembly is not yet complete because all the components are not yet coherent. Indeed, the *IndependentBank* component is not coherent because its *Manage_withdraw* composite port is not coherent. Another component is thus added to the assembly: the *BankIS* component is connected to the *IndependentBank* component through its *Request_Data* primitive port. At that point, the assembly is complete. One can consider that the removed component has been replaced by an assembly composed of the *IndependentBank* and the *BankIS* components[9].

3.4 Implementation and Experimentation

The two processes presented here (automatic component assembly building and dynamic substitution after a component removal) have both been implemented as an extension of the open-source Julia implementation[10] of the Fractal component model [3]. Our dynamic reconfiguration approach has been tested in the same environment we used to test the building process. To do so, we randomly generated the interfaces and ports of generated components, randomly choose functional objectives and then run the building process in order to build full complete assemblies [6]. For example, experiments were run with a library of 38

[9] In the example, it is a coincidence that the total number of removed components equals the number of components that are used to complete the assembly.

[10] `http://www.objectweb.org`

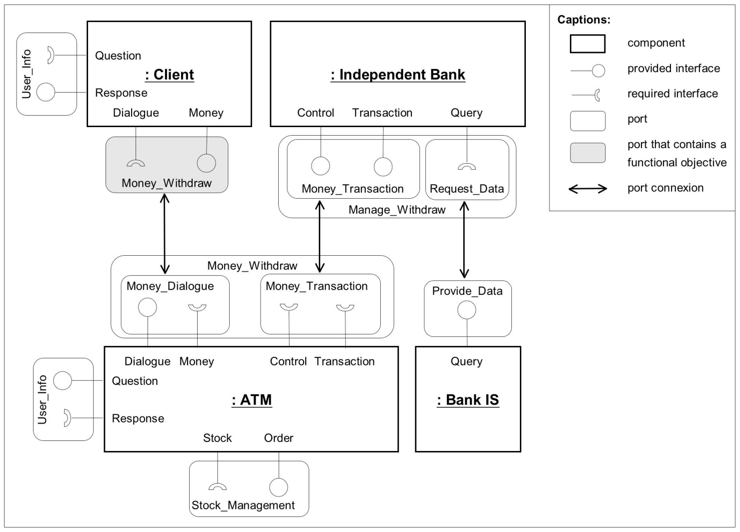


Fig. 5. Dynamic reconfiguration of the assembly

generated components. The search space contained more than 325 000 complete assemblies (complete search stopped after 15 hours). Among those complete assemblies, the largest ones have 48 connections and the smallest ones 18 connections. As a comparison, our optimized building algorithm finds the only minimal architecture composed of 7 connections in less than a second. To test our solution for evolution, a randomly chosen component was removed from a complete assembly and the substitution process was then triggered considering that the removed component was not available anymore. Those experiments showed that our solution provides alternative substitution possibilities (compared to existing one-to-one substitution mechanisms) thus is more flexible because it does not depend on the presence of a component that is able to exactly match the role of the removed one. In these experiments, in most cases, the result of substitution was a one-to-many substitution. We also noticed that the complexity of the mechanism exposed here is not higher than the complexity of the complete building process (which was efficient thanks to optimization strategies and heuristics).

4 Conclusion

The contribution of this paper is double. Firstly, we present an innovative solution for the dynamic replacement of a component from an assembly. This solution is not a component-to-component substitution but allows replacing a single component by a whole set of linked components while guaranteeing there is no functional regression. Secondly, we propose a property to identify useless components that can be removed. The advantage of this approach is that it can increase the number of reconfiguration possibilities by being less constraining. We implemented our solution as an extension of an existing open source implementation of the Fractal component model and successfully tested it on generated components.

The main limitations of this work is that we have not been able to try it on real components[11] but our experimentation framework allowed us to validate our ideas. Another limitation to our approach is that ports need to be added to the components in order to use them in our mechanisms. We believe this limitation is not very strong because ports can be provided by the component designer as an abstract view of the behavioral roles of the components that document the components, generated from protocols (in a design for reuse process) or abstracted from running assemblies that provide execution contexts for the components (in a design by reuse approach).

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
2. Plásil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: Proc. of the Int. Conf. on Configurable Distributed Systems, Washington, DC, USA, IEEE Computer Society (1998) 43–52
3. Bruneton, E., Coupaye, T., Stefani, J.: Fractal specification - v 2.0.3 (2004) `http://fractal.objectweb.org/specification/index.html`.
4. George, B., Fleurquin, R., Sadou, S.: A substitution model for software components. In: Proc. of the 2006 ECOOP Workshop on Quantitative Approaches on Object-Oriented Software Engineering (QaOOSE'06), Nantes, France (2006)
5. Bures, T., Hnetynka, P., Plásil, F.: Sofa 2.0: Balancing advanced features in a hierarchical component model. In: SERA, IEEE Computer Society (2006) 40–48
6. Desnos, N., Vauttier, S., Urtado, C., Huchard, M.: Automating the building of software component architectures. In Volker Gruhn, F.O., ed.: Software Architecture: 3^{rd} European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA). Volume 4344 of LNCS., Springer (2006) 228–235
7. Brown, A.W., Wallnau, K.C.: The current state of CBSE. IEEE Software **15**(5) (1998) 37–46
8. Crnkovic, I.: Component-based software engineering—new challenges in software development. Software Focus (2001)
9. Dijkman, R.M., Almeida, J.P.A., Quartel, D.A.: Verifying the correctness of component-based applications that support business processes. In Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K., eds.: Proc. of the 6th Workshop on CBSE: Automated Reasoning and Prediction, Portland, Oregon, USA (2003) 43–48
10. Medvidovic, N., N.Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1) (2000) 70–93

[11] It is not yet possible to find real component bases that are already documented with protocols. We believe that research work aiming at facilitating component reuse will encourage the building of such component repositories and provide us better experimentation frameworks in the future.

11. Inverardi, P., Wolf, A.L., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. ACM Trans. Softw. Eng. Methodol. **9**(3) (2000) 239–272
12. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. of the 8*th* European software engineering conference, New York, NY, USA, ACM Press (2001) 109–120
13. Mach, M., Plásil, F., Kofron, J.: Behavior protocols verification: Fighting state explosion. International Journal of Computer and Information Science (2005)
14. Wallnau, K.C.: Volume III: A technology for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon University, Pittsburgh, OH, USA (2003)
15. Adamek, J., Plásil, F.: Partial bindings of components - any harm? In: APSEC '04: Proc. of the 11th Asia-Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society (2004) 632–639
16. Reussner, R.H., Poernomo, I.H., Schmidt, H.W.: Reasoning on software architectures with contractually specified components. In Cechich, A., Piattini, M., Vallecillo, A., eds.: Component-Based Software Quality: Methods and Techniques. Volume 2693 of LNCS. Springer (2003) 287–325
17. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: SIGSOFT '96: Proc. of the 4th ACM SIGSOFT symposium on Foundations of software engineering, New York, NY, USA, ACM Press (1996) 3–14
18. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: Proc. of ICSE, Orlando, FL, USA, ACM Press (2002) 187–197
19. Inverardi, P., Wolf, A.L.: Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. Softw. Eng. **21**(4) (1995) 373–386
20. Allen, R.J.: A formal approach to software architecture. PhD thesis, Carnegie Mellon, School of Computer Science (1997) Issued as CMU Technical Report CMU-CS-97-144.
21. Matevska-Meyer, J., Hasselbring, W., Reussner., R.H.: A software architecture description supporting component deployment and system runtime reconfiguration. In: Proc. of the 9*th* Int. Workshop on Component-Oriented Programming (WCOP '04), Oslo, Norway (2004)
22. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Intl. Conf. on Software Engineering, Kyoto, Japan (1998)
23. Brada, P.: Component change and version identification in SOFA. In: SOFSEM '99: Proc. of the 26th Conf. on Current Trends in Theory and Practice of Informatics, London, UK, Springer-Verlag (1999) 360–368
24. Desnos, N., Urtado, C., Vauttier, S., Huchard, M.: Helping the architect build component-based architectures. In Rousseau, R., Urtado, C., Vauttier, S., eds.: Proc. of the 12*th* french speaking conference on Languages and Models with Objets (LMO2006), Nîmes, France, Hermès (2006) 37–52 (in french).
25. OMG: Unified modeling language: Superstructure, version 2.0 (2002) `http://www.omg.org/uml`.
26. Lobo, A.E., de C. Guerra, P.A., Filho, F.C., Rubira, C.M.F.: A systematic approach for the evolution of reusable software components. In: ECOOP'2005 Workshop on Architecture-Centric Evolution, Glasgow (2005)
27. Jones, C.: Systematic Software Development using VDM (2nd Edition). Prentice-Hall (1990)
28. Abrial, J.R.: The B-Book, Assigning programs to meanings. Cambridge University Press (1996)