Search-based many-to-one component substitution

Nicolas Desnos, Marianne Huchard, Guy Tremblay, Christelle Urtado, Sylvain Vauttier

Search-based many-to-one component substitution

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

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

SUMMARY

In this paper, we present a search-based automatic many-to-one component substitution mechanism. When a component is removed from an assembly to overcome component obsolescence, failure or unavailability, most existing systems perform component-to-component (one-to-one) substitution. Thus, they only handle situations where a specific candidate component is available. As this is not the most frequent case, it would be more flexible to allow a single component to be replaced by a whole component assembly (many-to-one component substitution). We propose such an automatic substitution mechanism, which does not require the possible changes to be anticipated and which preserves the quality of the assembly. This mechanism requires components to be enhanced with *ports*, which provide synthetic information on components' assembling capabilities. Such port-enhanced components then constitute input data for a search-based mechanism that looks for possible assemblies using various heuristics to tame complexity.

KEY WORDS: component substitution, component assembly evolution, search-based building process, many-to-one component replacement, heuristics, dead components

Introduction

Nowadays, software systems have to meet the needs of long life, autonomous and ubiquitous applications and must therefore be flexible, dynamic, and adaptable like never before.

[†]E-mail: Nicolas.Desnos@ema.fr, huchard@lirmm.fr, tremblay.guy@uqam.ca, Christelle.Urtado@ema.fr, Sylvain.Vauttier@ema.fr

Component-based software engineering (CBSE) [1] is a good solution to optimize software reuse and dynamic evolution while guaranteeing the quality of the software. 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) while enforcing a set of properties (its non-functional requirements) 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 long life applications or evolving environments, component substitution is a necessary mechanism for software evolution: it is a response to such events as component obsolescence, failure or unavailability. Anticipating valid component substitutions while designing some software is not always possible as the various contexts in which that software may run are not known in advance. Repairing a component assembly after a component has been removed while still preserving its whole set of functionalities is thus difficult. When a component is removed from an assembly, most existing approaches perform component-to-component (one-to-one) substitution [2, 3, 4, 5]. However, these approaches rely on the fact that an appropriate component, candidate for substitution, is available. This situation can hardly happen because it is difficult to find a component that has the same capabilities as 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 article, we propose an automatic substitution mechanism such that the possible changes do not need to be anticipated. Our approach uses primitive and composite ports for ensuring that a component can be replaced by a group of components while preserving the quality of the whole assembly. Such many-to-one component replacements are allowed by a search-based building algorithm that combines backtracking and branch and bound techniques to examine candidate assemblies. This algorithm is optimized using various search strategies and heuristics.

The rest of this paper proceeds as follows. The first two sections set up the context of this work. First, we briefly recall the typical CBSE process, in order to define correctness and completeness of an architecture. Then, we analyze the needs and limits of state-of-the-art practices for dynamic architecture reconfiguration. The following sections introduce our contribution. First, we present how ports allow us to automatically build valid assemblies [6] and how the assembly building process can be seen as a search-based problem, more precisely as a CSP. We then show how our building algorithm can be used as part of a four step component substitution process, and discuss how the complexity of the algorithm can be tamed using various search strategies and heuristics. Next, we discuss our implementation as well as some experiments we performed. Finally, we conclude and discuss some possible future work.

Software Architecture Correctness and Completeness in CBSE

This section discusses the issues of correctness and completeness of software architectures that result from a component reuse-based development process.

Typical CBSE process. Cbse [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 design validity is checked. An architecture is built to fulfill a set of functional objectives (its functional requirements) [8, 9]. Functional objectives are deduced from problem analysis and defined as a set of functionalities to be executed on selected components. Then, the structure of the architecture is described as a static interconnection of software component classes through their interfaces. This requires both selecting and connecting 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 behavioral dependencies between components. For simplicity's sake, this work only focuses on preserving functional requirements while the software evolves. Non-functional properties are also important but can be handled only after the functional constraints have been satisfied. Once the architecture is described, its validity is statically checked. Most systems verify the correctness of the architecture, while some also guarantee its completeness—both notions are described below. Once the validity of the architecture is checked, it can be deployed. 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. At runtime, the component assembly executes.

The evolution of this assembly is an important issue for the application to adapt to its environment in situations such as maintenance, evolution of the requirements, fault-tolerance, component unavailability, 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.

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 (interfaces, contracts, ports, etc.). The most precise checks are done by protocol comparison, which is a hard combinatorial problem [11, 12, 13, 14, 15].

Completeness. An architecture must guarantee that all its functional objectives will be met. In other words, the connections of an architecture must be sufficient to allow the execution of collaborations that reach (include) all its functional objectives. We call this **completeness** of the architecture [6]. Indeed, the use of a component functionality (modeled by the connection

[†]We assume that the selected components need no adaptation (or have already been adapted) as it is the only situation in which the components' external definitions are sufficient to *match* (whatever the definition of matching is) with other components' needs. Complementary approaches, interested in automating the assembly building process and performing component adaptation, must necessarily rely on additional information (*e.g.*, domain specific semantics, data or usage patterns) either provided by designers or collected through a reingeneering process. Thus, our approach is lighter.

of an interface) can require 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:

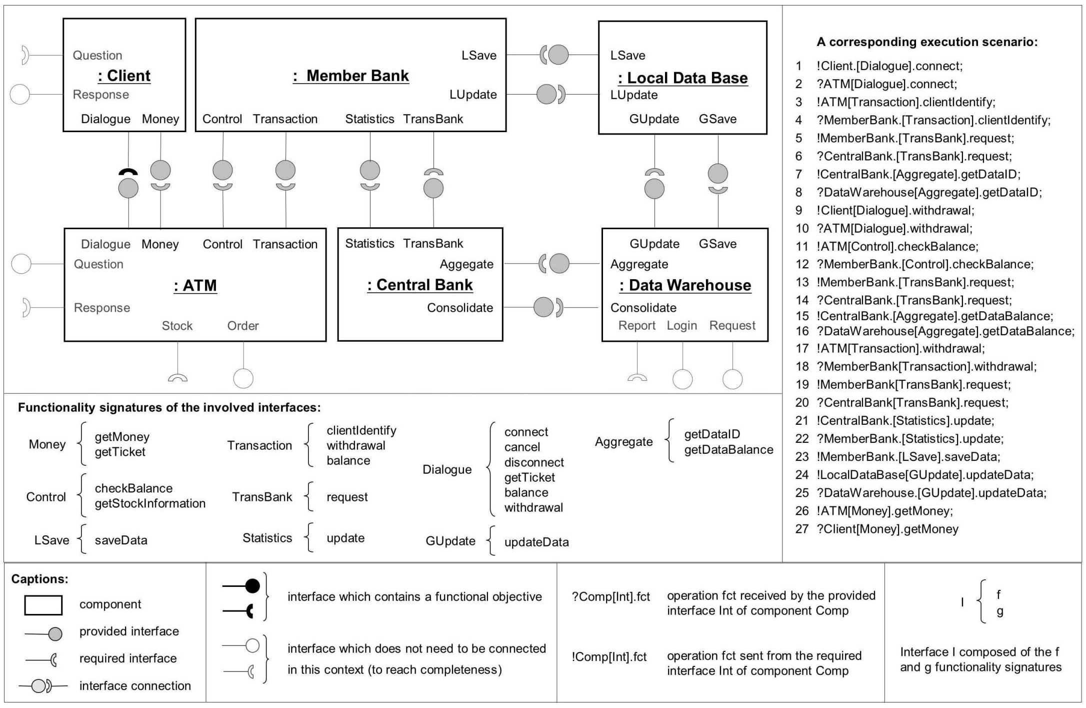
- For a first class of systems [16], completeness of an architecture is characterized by the fact that all interfaces of any of the architecture's component are connected. This notion of completeness is simple to check but over-constrains the assembly, thus diminishing both the capability of individual components to be reused in various contexts and the possibilities of building a complete architecture given a set of predefined components.
- A second class of systems [3] defines two categories of interfaces, namely, mandatory and optional. An architecture is then considered complete if all mandatory interfaces are connected, while optional ones can be left pending. This does not complicate completeness checking, yet 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, more relaxed view of completeness, requires connecting only the interfaces that are strictly necessary [12, 17, 18] by exploiting the component behavior's description. This is typically done by analyzing protocols which makes completeness checking less immediate.

In order to build correct and complete component assemblies we consider having as precise correction checking as possible and adopt the third vision on completeness while trying to limit the costs of protocol comparison by dismissing the less useful information. To achieve this, we define a port-enhanced component model.

Example. Figure 1 illustrates that completeness of an assembly can be ensured while connecting only the strictly necessary interfaces. For simplicity's sake, in the example, compatible operations and interfaces have the same name whereas, in the general case, interface types only need to be substitutable. The *Dialogue* interface from the *Client* component represents a functional objective and must therefore be connected. As can be deduced 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, as shown in line 12 of the execution scenario, the *Control* interface from the *MemberBank* component must be connected whereas the *Question* interface from the *Client* component, which is not used in the scenario, does not need to be connected.

Dynamic Architecture Reconfiguration

This section discusses correctness and completeness issues for evolving software assemblies. To ensure that an evolving valid component assembly remains valid at runtime, all systems try to control how the assembly evolves. Different evolution policies exist:


Question
: Client
Response
Dialogue
Money
: Member Bank
LSave
LUpdate
Control
Transaction
Statistics
TransBank
LSave
: Local Data Base
LUpdate
GUpdate
GSave
Dialogue
Money
Control
Transaction
Question
: ATM
Response
Stock
Order
Statistics
TransBank
Aggregate
: Central Bank
Consolidate
GUpdate
GSave
Aggregate
: Data Warehouse
Consolidate
Report
Login
Request
Functionality signatures of the involved interfaces:
Money { getMoney getTicket
Transaction { clientIdentify withdrawal balance
Dialogue { connect cancel disconnect getTicket balance withdrawal
Aggregate { getDataID getDataBalance
Control { checkBalance getStockInformation
TransBank { request
LSave { saveData
Statistics { update
GUpdate { updateData
Captions:
component
provided interface
required interface
interface connection
interface which contains a functional objective
interface which does not need to be connected in this context (to reach completeness)
?Comp[Int].fct
operation fct received by the provided interface Int of component Comp
!Comp[Int].fct
operation fct sent from the required interface Int of component Comp
I { f g
Interface I composed of the f and g functionality signatures

```
A corresponding execution scenario:
1  !Client.[Dialogue].connect;
2  ?ATM[Dialogue].connect;
3  !ATM[Transaction].clientIdentify;
4  ?MemberBank.[Transaction].clientIdentify;
5  !MemberBank.[TransBank].request;
6  ?CentralBank.[TransBank].request;
7  !CentralBank.[Aggregate].getDataID;
8  ?DataWarehouse[Aggregate].getDataID;
9  !Client[Dialogue].withdrawal;
10 ?ATM[Dialogue].withdrawal;
11 !ATM[Control].checkBalance;
12 ?MemberBank.[Control].checkBalance;
13 !MemberBank.[TransBank].request;
14 ?CentralBank.[TransBank].request;
15 !CentralBank.[Aggregate].getDataBalance;
16 ?DataWarehouse[Aggregate].getDataBalance;
17 !ATM[Transaction].withdrawal;
18 ?MemberBank[Transaction].withdrawal;
19 !MemberBank[TransBank].request;
20 ?CentralBank[TransBank].request;
21 !CentralBank.[Statistics].update;
22 ?MemberBank.[Statistics].update;
23 !MemberBank.[LSave].saveData;
24 !LocalDataBase[GUpdate].updateData;
25 ?DataWarehouse.[GUpdate].updateData;
26 !ATM[Money].getMoney;
27 ?Client[Money].getMoney
```

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

- Some systems simply forbid dynamic reconfigurations [19, 20], so assemblies cannot evolve at runtime. This, of course, is an unsatisfactory policy.
- Some systems [21, 3] allow the architectural structure to be violated when modifying component assemblies at runtime. They allow 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.
- Other systems ensure that component assemblies always conform to the architectural structure. All possible evolutions must therefore be anticipated at design-time and described in the architecture itself [10]. Different techniques can be used. For example, [22, 5] use patterns to specify which interfaces can be connected or disconnected and which components can be added or removed. [23, 24] use logical rules, a more powerful means to describe the possible evolutions. These solutions, however, complicate the design process and make anticipation necessary, which is not always possible [5, 25].

Dynamic Component Removal. Among the situations that must be handled to enable component assembly evolution is dynamic component removal. Other facets of interest in the change process are related to identifying changes, interrupting components' execution, saving the removed components' state, deploying the new components, and migrating the saved states to the new components[‡]. When removing a component from an assembly, the main constraint is to ensure that there will be no functional regression. The third category of systems mentioned above typically allow a removed component to be replaced by a component which provides compatible services in order for the assembly to still conform to the architecture. Anticipating possible evolutions allows these systems to ensure that the new assembly will still be valid, as it has been statically checked on the architecture at design-time.

There are two major interpretations of component compatibility. In most systems [26, 2, 5, 3], components must strictly be compatible: the new component must provide at least the provided interfaces of the removed component and cannot require more required interfaces. In [27], 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 necessary in this context). On the other hand, the new component can have additional required interfaces as soon as they find a compatible provided interface among the assembly's components. This context-dependent definition of compatibility allows better adaptability of assemblies.

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

1. Anticipating all possible evolutions to include their description in the initial, design-time, architecture description is not always possible because it requires knowing all situations that may occur in the system's future evolution. Ideally, it would be better to 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 generally unlikely that a component having compatible interfaces exist among the potential candidates for substitution. Yet, in the (more frequent) case when an adequate component does not exist, it might be possible to replace the removed component by a set of linked components that, together, provide the required services.

The problem of replacing a removed component by an assembly of components in an unanticipated way while guaranteeing, as much as possible, the quality (executability) of the whole assembly is the initial motivation for the work presented in this paper.

[‡]Even though we have not yet studied the deployment process itself, our system could help identify which components might be impacted when removing some components, thus minimizing the number of components that need to be interrupted. Moreover, as far as state consistency is concerned, we assume, as in all Cbse approaches, that no assumption can be made on the components' implementation. This assumption thus makes state migration impossible, as it would constrain the internal structure of components. If some change occurs, a robust implementation of our sytem would rollback the states of all components so that they all are in some initial state that enables them to be restarted safely.

Port Enhanced Components for Incremental Assembly Completeness Checking

This section describes how adding ports to components provides a means to automatically build complete component assemblies [6, 28]. Existing approaches usually statically describe architectures in a top-down manner. Once the architecture is defined, they verify its validity using costly checking algorithms [11, 12, 13, 14, 15]. 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 require. To reduce the complexity, we chose to simplify the information contained in protocols—more precisely, behavior protocols such as those used in SOFA [29], which are regular expressions that express the various possible sequences of events (traces) allowed by a component—and represent this information 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 underlying idea for building a complete component assembly is to start from the functional objectives, select the suitable components, and then establish the necessary links between them. Completeness is a global property that we will 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 thus enhance the component model with the notion of port, to model the information that is strictly necessary to guarantee completeness in an abstract way. Primitive and composite ports will 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. Primitive ports are used to model simple usage contexts (possible collaboration between two components) and composed into composite ports that model more complex contexts (complex collaborations). As in UML 2.0 [30], 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 a component that enables the component to play a precise role with respect to a given use case.

Primitive ports are composed of interfaces, as in many other component models [30, 31]. 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: here, a collaboration the *Client* component can establish with another (yet unknown) component to withdraw money. Connecting two primitive ports is an atomic operation that connects their interfaces: two primitive ports are connected together when all

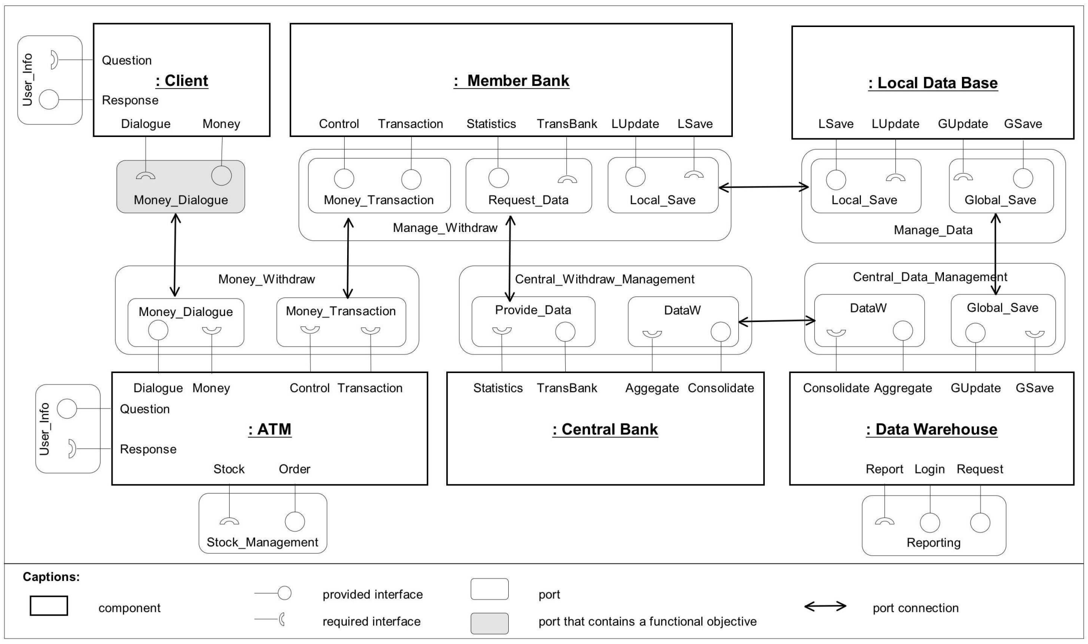


Figure 2. Example of components with primitive and composite ports

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. They model complex collaborations that are composed of finer grained ones (modeled by the sub-ports). Indeed, they provide an abstract description of a part of the component behavior—less information than in component behavior protocols but more information than the syntactic description of component capabiliies modelled by interfaces. 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 named *Money_Withdraw* which is composed of the *Money_Dialogue* and *Money_Transaction* primitive ports.

Much like a designer must 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 that, if true, guarantees that the component is adequately connected to its immediate neighbors, and then to aggregate these local values into a global completeness property. We call this local property **coherence** and have shown [6] that it is a necessary condition for validity. Intuitively, we will see that when all components of an assembly are coherent, the assembly is complete. A component is said to be coherent if all its exposed (top-level) composite ports are and these latter are coherent if their primitive ports are connected in a coherent way (see below).

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 port with another unrelated composite port and when it does share some primitive ports.

An exposed composite port is said to be coherent if one of these three mutually exclusive cases holds:

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, the composite port can still be coherent if it shares the connected primitive ports 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 long as its shared primitive ports are connected in the context of another (significant) composite port.

A component is said to be coherent if all its exposed composite ports are 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.

In the next section, we provide more formal definitions in order to show that the building of all complete component assemblies can be seen as a search-based problem.

Building Complete Component Assemblies: a Search-Based Problem

Formal Definition of Completeness. More formally, completeness can be described after setting some preliminary definitions.

- We define a **component** C as a quintuple:

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

  Prv_C is the set of C's provided interfaces and Req_C its set of required interfaces. $Prim_C$ is the set of all C's primitive ports, $Comp_C$ its whole set of composite ports and $TopComp_C \subseteq Comp_C$ its set of exposed (top-level) composite ports.

- We denote by $\mathit{Int}_C = \mathit{Prv}_C \cup \mathit{Req}_C$ the whole set of C's interfaces and by $\mathit{Ports}_C = \mathit{Prim}_C \cup \mathit{Comp}_C$ the whole set of C's ports.
- An **interface** is characterized by a set of operation signatures (its interface type) and a direction (provided or required). We assume, as in most object-oriented languages (*e.g.*, Java) and modeling languages (*e.g.*, UML), that interface types are partially ordered in a specialization hierarchy. If not, or if a finer definition is required, it is always possible to (re)define such a specialization relation as we have done in [32].
- A **primitive port** ρ is a set of interfaces. Let $\rho \in \mathit{Prim}_C$ be a primitive port of C, $\rho \subseteq \mathit{Int}_C$.
- A **composite port** γ of C is a set of ports, primitive or composite, from C, subject to some restrictions described below.
- Let $\gamma \in \mathit{Comp}_C$ be a composite port of C, where $\gamma \in 2^{\mathit{Ports}_C}$. We define $\mathit{PrimPorts}^*(\gamma)$, resp. $\mathit{CompPorts}^*(\gamma)$, as the set of all primitive, resp. composite, ports that are directly or indirectly contained in γ:

$$\begin{aligned} \mathit{PrimPorts}^*(\gamma) &= \{\rho \in \gamma \cap \mathit{Prim}_C\} \cup \bigcup_{\gamma' \in \gamma \cap \mathit{Comp}_C} \mathit{PrimPorts}^*(\gamma') \\ \mathit{CompPorts}^*(\gamma) &= \{\gamma' \in \gamma \cap \mathit{Comp}_C\} \cup \bigcup_{\gamma' \in \gamma \cap \mathit{Comp}_C} \mathit{CompPorts}^*(\gamma') \end{aligned}$$

  Note that for γ to be well-defined, γ cannot be a (direct or indirect) sub-port of itself, that is, $\gamma \notin \mathit{CompPorts}^*(\gamma)$.
  For component C to be well-defined, each of its composite ports must either itself be or be a sub-port of an exposed composite port of C. This can be expressed as:

$$\forall \gamma \in \mathit{Comp}_C \cdot \gamma \in \mathit{TopComp}_C \ \vee \ \exists \gamma' \in \mathit{TopComp}_C \cdot \gamma \in \mathit{CompPorts}^*(\gamma')$$

- Let i be an interface. We denote by $\mathit{Dir}(i) \in \{\mathit{pro}, \mathit{req}\}$ the direction of interface i and by $\mathit{Type}(i)$ its type. We denote by $\preceq$ the specialization relation between interface types. An interface i is said to be **compatible** with an interface i' iff the provided interface type is equal to or more specific than the required interface type:

$$\mathit{Compat}(i, i') = \oplus \begin{cases} \mathit{Dir}(i) = \mathit{pro} \wedge \mathit{Dir}(i') = \mathit{req} \wedge \mathit{Type}(i) \preceq \mathit{Type}(i') \\ \mathit{Dir}(i) = \mathit{req} \wedge \mathit{Dir}(i') = \mathit{pro} \wedge \mathit{Type}(i') \preceq \mathit{Type}(i) \end{cases}$$

- A primitive port ρ is said to be **compatible** with another primitive port ρ', noted $(\rho, \rho') \in \mathcal{R}_{comp}$, iff there is a bijection from one's set of interfaces to the other's set of interfaces such that corresponding interfaces are compatible. Primitive port compatibility is symmetric.

$$(\rho, \rho') \in \mathcal{R}_{comp} = \exists f : \rho \rightarrow \rho' \cdot \forall i' \in \rho' \cdot \exists!\, i \in \rho \cdot f(i) = i' \wedge \mathit{Compat}(i, i')$$

Let us now consider a component assembly that involves a set of components and a set of primitive port connections.

- 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.

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

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

- Let $\gamma \in \mathit{TopComp}_C$ be a top-level composite port of component C. $\mathit{Shared}_C(\gamma)$ is the set of primitive ports shared by γ and by any other top-level composite port of C:

$$\mathit{Shared}_C(\gamma) = \{\rho \in \mathit{PrimPorts}^*(\gamma) \mid \exists\, \gamma' \in \mathit{TopComp}_C \cdot \gamma \neq \gamma' \wedge \rho \in \mathit{PrimPorts}^*(\gamma')\}$$

Given an exposed composite port $\gamma \in \mathit{TopComp}_C$, three mutually exclusive cases are possible for γ to be coherent as argued in the previous section.

- $\gamma \in \mathit{TopComp}_C$ is **coherent** (with respect to component C) if the following holds:

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

- A component C is **coherent** iff: $\forall\, \gamma \in \mathit{TopComp}_C \cdot \gamma$ is coherent

Building All Complete Assemblies as a Constraint Satisfaction Problem. The inputs of our problem are:

- A component repository. This repository is characterized by the set $\mathit{\Pi}$ of all primitive ports from all the components in the repository, and by the set $\mathit{TopComp}$ of all the exposed (top-level) composite ports from all the components in the repository.
- Functional objectives. These functional objectives are defined through $\mathcal{O} \subseteq \mathit{\Pi}$, the set of primitive ports which match the functional objectives.

Let us now define $\mathit{Role}(\rho)$ as the set of all the exposed composite ports to which a primitive port ρ belongs.

$$\mathit{Role}(\rho) = \{\gamma \in \mathit{TopComp} \mid \rho \in \mathit{PrimPorts}^*(\gamma)\}$$

We also note $\mathit{Compatible}(\rho)$ the set of all primitive ports in $\mathit{\Pi}$ that are compatible with a primitive port ρ.

Let $\mathit{Connections}_\rho = \{x^\rho_\gamma\}_{\rho \in \mathit{\Pi}, \gamma \in \mathit{Role}(\rho)}$ be the set of variables that represent the connections of a primitive port ρ.

Each variable represents the connection of a primitive port in the context of one of the exposed composite ports it belongs to. The connection of a shared primitive port is thus represented by several variables. Each variable enables to distinguish the different connection contexts, in which a shared primitive port is considered at the same time as connected, when

[§]As in VDM [33] and B [34], "·" separates the (typed) variable introduced by the quantifier and the associated predicate.

it participates to the connection of a connected exposed composite port, or unconnected, when it belongs to another unconnected exposed composite port.

Let $\mathit{Connections} = \bigcup_{\rho \in \Pi} \mathit{Connections}_\rho$ be the set of variables that are used to describe all connections between all existing components. *Connections* is thus the set of variables of the Csp we have to solve. The value domains of these variables are:

$$\forall\, x_\gamma^\rho \in \mathit{Connections} \cdot \mathit{Dom}(x_\gamma^\rho) = \mathit{Compatible}(\rho) \cup \{\mathit{nil}\}$$

Given those value domains, each variable, which represents a given primitive port, can be assigned as value the primitive port to which it is connected—*nil* is a special value indicating that the primitive port is unconnected.

Building a component assembly then amounts to assigning values for the various variables of *Connections*, with respect to a set of constraints that guarantee the consistency of the architecture:

1. *Constraints on functional objectives.* All primitive ports selected as functional objectives must be connected in the solution.

$$\forall\, \rho \in \mathcal{O} \cdot \exists\, x_\gamma^\rho \cdot x_\gamma^\rho \neq \mathit{nil}$$

2. *Constraints on port connection symmetry.* When a primitive port is connected to another primitive port, then the latter primitive port must be connected to the former.

$$x_\gamma^\rho = \rho' \Rightarrow \exists\, x_{\gamma'}^{\rho'} \cdot x_{\gamma'}^{\rho'} = \rho$$

3. *Constraints on exposed composite port coherence.* The variables that correspond to connections of primitive ports of an exposed composite port must either all be set to *nil* or all be set to some *non-nil* value.

$$\forall\, \gamma \in \mathit{TopComp} \cdot \forall\, \rho, \rho' \in \mathit{PrimPorts}^*(\gamma) \cdot x_\gamma^\rho \neq \mathit{nil} \Rightarrow x_\gamma^{\rho'} \neq \mathit{nil} \;\oplus\; x_\gamma^\rho = \mathit{nil} \Rightarrow x_\gamma^{\rho'} = \mathit{nil}$$

4. *Constraints on shared primitive port connection well-formedness.* When a shared primitive port belongs to several connected exposed composite ports, it must be connected to the same primitive port in every context.

$$\forall\, \gamma, \gamma' \in \mathit{TopComp} \cdot \forall\, \rho \in \mathit{Shared}_C(\gamma) \cap \mathit{Shared}_C(\gamma') \cdot x_\gamma^\rho \neq \mathit{nil} \;\wedge\; x_{\gamma'}^\rho \neq \mathit{nil} \Rightarrow x_\gamma^\rho = x_{\gamma'}^\rho$$

When there is no functional objective, a trivial solution that satisfies all the constraints is an assembly with no connection (every variable in *Connections* is *nil*). Every defined functional objective adds a constraint that excludes *nil* from the domain of the associated variable: the corresponding port must be connected. A non-trivial solution must then be found thanks to a combination of different problem solving techniques. We propose a backtracking algorithm that enumerates the possible variable assignment combinations, optimized with strategies that prune the search tree and heuristics that speed up its traversal—thus effectively resulting in a branch-and-bound strategy. These search techniques are combined with constraint propagation (arc consistency) that filter inconsistent values from variable domains to reduce the search space. This algorithm, its optimizations and its results are presented next.

Overview of the Incremental Building Process. The principle of our automatic building process is first to connect all the primitive ports representing functional objectives and then to iteratively list and connect all the primitive ports that must be connected to maintain the coherence of the components' exposed composite ports. This process is implemented as a depth-first traversal of a construction tree. Backtracking allows a complete exploration of every construction paths (alternative connection choices), thus ensuring that all possible solutions are found.

The building algorithm uses a set (FO-set) that always contains a list of the ports that still have to be connected. This FO-set contains only primitive ports: when a composite port (γ) has to be connected, it is decomposed into the set of primitive ports it contains, directly or indirectly ($\mathit{PrimPorts}^*(\gamma)$), and these primitive ports are added to the FO-set. The FO-set is initialized with the primitive ports that correspond to the functional objectives. The building process can be decomposed into three steps:

1. *Choice of the primitive port.* One of the primitive ports is selected from the FO-set.
2. *Choice of a compatible unconnected primitive port and connection.* Compatible primitive ports are searched for amongst the ports of components from the repository or from the already built sub-assembly. If compatible unconnected ports are found, one of them is selected. If the chosen port belongs to a component that does not yet belong to the assembly, the component is added to the assembly. The two ports are then connected together.
3. *Choice of a collaboration context and update of the dependency set.* If the chosen compatible port belongs to a single exposed composite port, all other primitive ports of that composite port are added to the FO-set. If the chosen compatible port is shared by several exposed composite ports, one of those exposed composite ports (defining one of the possible collaboration contexts) is chosen as a collaboration context and its primitive ports are added to the FO-set. The other exposed composite ports may in turn be chosen when the building process backtracks to explore another solution. In any case, no port dependencies—and therefore no interface dependencies—are left unsatisfied.

These three steps are iterated until the FO-set is empty. All the initial primitive ports that represent functional objectives are then connected along with all ports they are recursively dependent upon: the resulting assembly is thus complete. During the whole process, backtracking allows to both rollback unsuccessful connection attempts—past connection choices lead to a situation where there is no available primitive port where to connect a primitive port from the FO-set—and build all possible complete assemblies. This enumerative building process, of which the basic principle is presented here, is highly combinatorial. Optimization strategies and heuristics have been used to speed up the traversal of the construction space as presented below.

As a result, the building algorithm provides a set of complete architectures. Since architecture completeness is a necessary condition for architecture validity, the resulting set of complete architectures provides preselected assemblies on which classical correctness checkers, such as [5], can then be used.

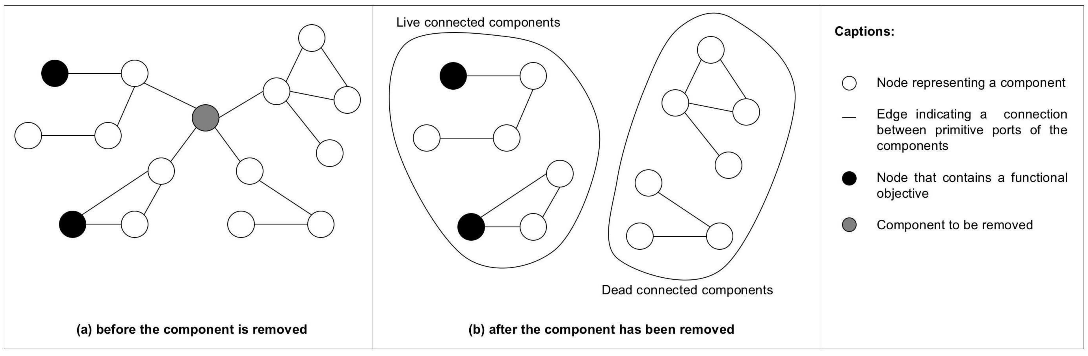

Live connected components
Dead connected components
(a) before the component is removed
(b) after the component has been removed
Captions:
Node representing a component
Edge indicating a connection between primitive ports of the components
Node that contains a functional objective
Component to be removed

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

Many-to-one 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 Dead Components. When a component has been removed from a complete assembly, some parts of the assembly may 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:

$$A = (G_A, FO_A)$$

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[¶].

If we consider the graph that results from the removal of the node representing the removed component, we can partition the graph in two parts: the connected components[‖] that have at least a node which contains a functional objective and the connected components without any node that contains a functional objective. The second part of the graph is no longer useful because the associated components were not in the assembly to fulfill some functional objectives but rather to fulfill some of the removed component's needs. 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.

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, all components which do not participate anymore in the assembly's completeness can be removed. Components from the dead connected components set $\mathcal{D}_{A,C}$ can be removed from the assembly because they only participated in the removed component's coherence. Indeed, as dependencies are modelled by edges of the graph, if there are unconnected subgraphs that are not needed to implement the functional objectives (which we call subgraphs of dead components), these subgraphs are useless (no dependency links them to the parts of the graphs that contain functional objectives).

Removing the dead components is a necessary step because keeping useless components add useless dependencies that make the resulting assembly considerably larger, 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 considering future improvements that would exploit the protocols to improve the detection of dead components.

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

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

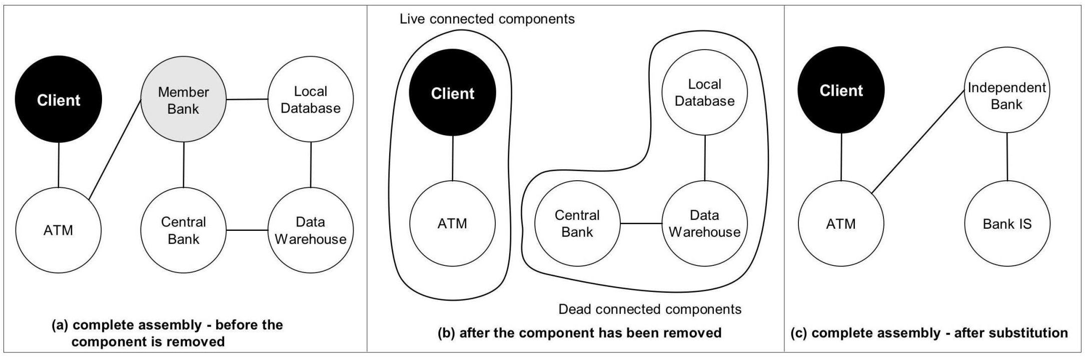

Client
Member Bank
Local Database
ATM
Central Bank
Data Warehouse
(a) complete assembly - before the component is removed
Live connected components
Client
Local Database
ATM
Central Bank
Data Warehouse
Dead connected components
(b) after the component has been removed
Client
Independent Bank
ATM
Bank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

Figure 4. Evolution scenario on the ATM example: removal of the *MemberBank* component

Re-building the Incomplete Assembly. Once the dead components have been removed from the component assembly, the assembly contains all the components necessary to ensure completeness except for one component (the removed one) along with its dependent components. Some of the remaining components' dependencies are not yet satisfied. The goal is then to find a single component (like other systems do) or a series of assembled components that can fulfill the same unsatisfied dependencies as the removed component did. We suppose 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) to have the possibility of replacing the removed component by a set of assembled components that, together, can replace the removed component.

The partial assembly in $\mathcal{L}_{A,C}$ is the re-building process starting point. It is considered an intermediate result of the global building process described above. The partial assembly is not complete yet: there still exist unsatisfied dependencies that were previously fulfilled by the removed component. These dependencies are identified, and the building process we described above is run to complete the architecture. In this case, the initial FO-set contains the primitive ports that correspond to unsatisfied dependencies, to which is added, if applicable, the removed component's primitive ports that were part of the assembly's initial functional objectives.

Evolution Scenario. For 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*, *CentralBank*, *LocalDatabase* and *DataWarehouse*) represent components which do not contain any functional objective. We also assume there is a component repository, which will be searched for possible replacement components. 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 three now consequently dead components ($\mathcal{D}_{ATMexample,MemberBank} = \{CentralBank, LocalDatabase, DataWarehouse\}$)

have been removed. To complete the assembly, new components must be added. Figure 4(c) sketches the result of this re-building process: The *IndependentBank* component 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, *MemberBank* is the component to remove. When it is removed, completeness of the architecture is lost. Indeed, the *ATM* component is not locally coherent anymore. Its *Money_Withdraw* composite port is not coherent because the primitive port *Money_Transaction* is not connected while the *Money_Dialogue* primitive port is not shared and still connected. The *CentralBank*, *LocalDatabase* and *DataWarehouse* components constitute the $\mathcal{D}_{ATMexample,MemberBank}$ graph and can also be removed. Completeness is reached 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 this point, the assembly is complete. As a result, one can then consider that the removed component has been replaced by an assembly composed of the *IndependentBank* and the *BankIS* components.

Optimization of the Re-building Process using Strategies and Heuristics

As described previously, the optimization problem is defined as a Csp. Our search space is the set of possible assemblies, considering only syntactical compatibility rules to connect ports. Assemblies that satisfy a set of functional objectives and consistency properties (connection dependencies) are searched for in this search space. Our solution strategy classically uses backtracking [35] to enumerate all possible connections and incrementally build all possible assemblies. Backtracking is combined with a branch-and-bound (B&B) strategy [36] that prunes the solution exploration tree. The objective function to be minimized is the number of connections in the assembly. For a given assembly, this amounts to minimizing the number of *non-nil* valuations for the $Connections_p$ variables. As quoted in [37], B&B techniques have little been used in Sbse although there are some exceptions: B&B is used to deal with *the next release problem* where requirements are chosen under some resource and dependency constraints [38], and for solving *the staffing problem* expressed as a Csp.

We measured the performance of the building algorithm. We chose to test the whole building algorithm instead of its restriction to the re-building of an incomplete assembly after the removal of a component. In other words, we started building assemblies from scratch instead of starting from an incomplete sub-assembly. For this purpose, we implemented a test environment that generates random component sets, thus providing various building contexts, differing in both size and complexity. Once a component set is generated, an arbitrary number of ports can be chosen as functional objectives and the building algorithm can be launched. Our experiments show that the combinatorial complexity of the building process is quite high, as illustrated in the next section. To use our approach in highly demanding situations, such

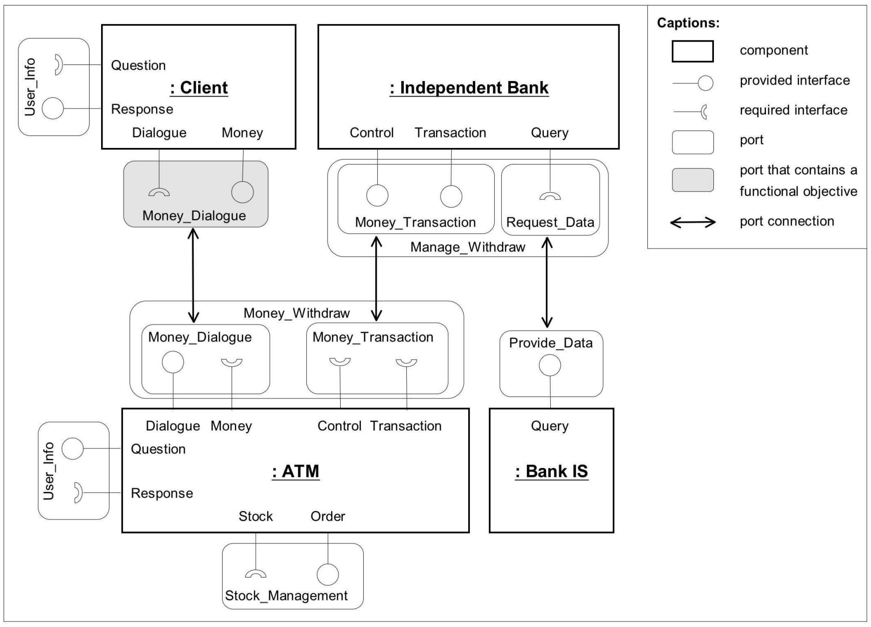


Figure 5. Dynamic reconfiguration of the assembly

as runtime deployment and configuration of components, we studied various heuristics that speed up the building process.

B&B strategy to build minimal assemblies. A first strategy is to try to find not all the possible assemblies but only the most interesting ones. Minimality is an interesting quality metrics for an assembly [39]. More precisely, we try to minimize the number of primitive port connections: fewer connections entail fewer semantic verifications, fewer interactions and, therefore, fewer conflict risks. Fewer connections also entail more evolution capabilities (free ports). To efficiently search for minimal assemblies, we added a branch-and-bound strategy to our building algorithm. The bound is the maximum number of primitive port connections allowed for the construction of the assembly. When this maximum is reached while exploring a branch of the construction tree, the rest of the branch can be discarded as any new solution will be suboptimal relative to any previously found solution (pruning).

Look-ahead (LA) Strategy. An estimate can be used to predict if traversing the current construction branch can lead to a minimal solution. This estimate is based on the minimum number of primitive port connections required to complete the building process. As soon as the sum of the estimate and the number of already existing connections is larger than the current bound, the branch can be pruned. A simple example of such an estimate is the number of ports in the FO-set divided by two, which corresponds to a lower bound of the number of connections needed to connect the ports that already are in the FO-set (in the most optimistic case, each port from the FO-set can connect to another port from the FO-set thus adding no new dependency and, moreover, satisfying two dependencies at once). A more selective and realistic estimate consists in calculating how many of primitive port pairs from the FO-set can be connected with one another. The number of remaining connections is higher than the cardinality of the FO-set minus the number of primitive port pairs.

Min Domain (MD) Heuristic. This heuristic is used to efficiently choose primitive ports from the FO-set in step 1 of the building algorithm. To each port can be associated the set of primitive ports it can be connected to. Amongst these, the port with the fewest free compatible ports is chosen first. This minimizes the effort to try all the connection possibilities: in case of repeated failures, impossible constructions can be detected sooner.

Min Effort (ME) Heuristic. In the branch-and-bound strategy, every time the bound is lowered, traversal of the tree is speeded up. During step 2 of the building algorithm, when choosing the compatible primitive port to connect to, the free compatible primitive port that belongs to the composite port (γ) that contains the fewest primitive ports (smallest $\mathit{Card}(\mathit{PrimPorts}^*(\gamma))$) is chosen first. This corresponds to choosing the primitive port that adds the fewest dependencies, thus minimizing future connection efforts. Another similar situation occurs during step 3 of the building algorithm: when the primitive port to connect to is chosen, if it is shared by several composite ports, then the composite port that contains the fewest primitive ports is chosen as the first collaboration context to explore.

No New Dependency (NND) Heuristic. In step 2 of the building algorithm, compatible ports are first searched for in the FO-set itself. When a compatible port can be found in that set, it is preferred to others because its connection will add no new dependency and, furthermore, will satisfy two dependencies at once—indeed, when a port belongs to the FO-set, its related primitive port is already in the FO-set.

Implementation and Experimentation

The two processes presented above (automatic component assembly building and dynamic substitution after a component removal) have both been implemented as an extension of the open-source Julia implementation[**] of the Fractal component model [3].

Experimentation framework. To evaluate the applicability and usefulness of the built assemblies and the optimization techniques, we needed a test environment. We were not able to experiment on real components because real-world component repositories, with properly documented behavior, are not yet available. Indeed, to (manually or semi-automatically) add

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

	Build 1	Build 2	Build 3	Subst 1
Number of components in a base	15	20	30	40
Max. number of primitive ports by component	10	10	10	29
Max. number of composite ports by component	4	10	10	3
Max. number of primitive ports by composite port	5	6	6	6

Table I. Variable values defining experimentation contexts: three building contexts of growing complexity and a substitution context

ports to components, component behavior must be described in an abstract way (for example, with protocols). We expect that research aiming at facilitating component reuse will encourage the building of such component repositories, thus providing better frameworks for future experimentation. To overcome this lack of real repositories, instead we simulated component repositories, aiming to define components as complex as real ones. Moreover, as a meantime alternative, we also plan to contribute to standardizing benchmarks in SBSE by providing our simulated repositories data online. As it is already the case in other applications of search-based methods, this will contribute to enabling comparisons and increasing reproducibility.

We implemented a test environment that generates random component sets, thus providing various building contexts, differing in both size and complexity. Once a component set is generated, an arbitrary number of ports can be chosen as functional objectives and the building algorithm can be launched. In this environment, a test repository has the following characteristics:

- *Fixed parameters.* The number of randomly generated method names is set to 5000, the number of randomly generated interfaces to 150, the maximum number of methods in an interface to 5, the maximum number of interfaces in a primitive port to 5, and the number of initial functional objectives to 3.
- *Variable parameters.* Depending on the experiments, we tried various values for some of the other characteristics. For example, the number of components in a component repository, the maximum number of primitive or composite ports by component, and the maximum number of primitive ports by composite port were variable parameters. This allowed us to have problem instances of various complexities.

Evaluation of the building algorithm. To evaluate the building algorithm, we empirically defined 3 building contexts that allowed to increase complexity and see how robust our heuristics were (see Table I). More precisely, for each context, we generated 3 different component repositories, and for each repository, we randomly chose 3 different initial functional objective sets. Then, for each FO-set, we ran the building algorithm, to build minimal complete assemblies, 5 times. Results are synthesized in Table II which shows how the algorithm behaves when various sets of strategies and heuristics are used. The table records both the percentage of runs that did not exceed 2700 seconds (45 minutes), and, when applicable, the average execution times (in seconds) for such runs. A run is a complete search for minimal solutions.

	No heuristic		B&B		B&B+LA		B&B+LA+MD		B&B+LA+ MD+NND+ME	
Build 1	55%	1347	100%	8	100%	2	100%	2	100%	1
Build 2	0%		0%		89%	22	100%	57	100%	9
Build 3	0%		16%	106	100%	12	100%	5	100%	3

Table II. Comparison of the percentage of completed runs and average execution time of the building algorithm while varying strategies and heuristics

% solved cases	80
% one-to-one substitution among solved cases	19
% reused dead components	20

Table III. Synthetic view of results for reconfiguration experiments

As execution is interrupted after 45 minutes, 0% means that all runs have been interrupted before the search for minimal solution was completed. 100% means that all runs succeeded in founding all minimal solutions. Execution times lower than one second are simply noted as 1 second. Results show how the whole set of strategies and heuristics are necessary for and efficient at taming the building process complexity. Minimal solutions vary in size from 3 to 35 connections and from 4 to 18 components. As the simulated situations corresponding to the third context seem to be more complex than any typical component assembly found in the literature, we decided it was not worth trying further heuristics or switching to an incomplete search method.

Evaluation of the dynamic reconfiguration approach. Our dynamic reconfiguration approach has been tested in the same environment used to test the building process. The experimentation context is defined by the variable values shown in last column of Table I. We generated 10 component repositories for this context. Then, to test our solution for evolution, we started from a generated complete component assembly from which a randomly chosen component was removed. The substitution process was then triggered by considering that the removed component was not available anymore. We ran the dynamic reconfiguration process 40 times, each time with a newly generated assembly (varying the set of functional objectives) and a new component to remove. Results are synthesized in Table III. 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 can exactly match (the role of) the removed one. In some situations (20%), no solution exists—the repository does not contain components that can be combined to be substituted to the removed one— but among the solved situations, 81% are solved thanks our many-to-one substitution proposal (compared to only 19% that can be solved with usual one-to-one substitution techniques). Furthermore, the resulting substitution was usually many-to-one. Also, we 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 the optimization strategies and heuristics.

Conclusion

To strengthen the ability of component-based software to dynamically evolve, we presented a solution for the dynamic replacement of a component from an assembly. Its originality lies in the fact that it is not restricted to component-to-component (one-to-one) substitution. Our approach requires that components carry information on the possible collaborations they can establish with other components, embodied by primitive and composite ports (similar to complex plugs). Using this information, a search-based mechanism builds a minimal sub-assembly in order to replace the removed component while guaranteeing there is no functional regression. A cleaning step then removes the useless components. The advantage of this approach is that it increases the number of reconfiguration possibilities by being less constraining. As the problem of assembly (re-)building is highly combinatorial, optimization strategies and heuristics have been proposed, implemented, and compared. The whole solution is implemented as an extension of an existing open source implementation of the Fractal component model and successfully tested on generated components.

Next steps will consist in experimenting our approach using real-world software components: our experimentation framework allowed us to validate our approach and be confident that it can deal with realistic situations. Another open issue is component documentation with primitive and composite ports. We are currently investigating strategies to automatically generate ports from protocols (in a design for reuse process) or from execution traces obtained by executing component assemblies (in a design by reuse approach). Run-time replacement of a component also raises the problem of identifying the minimal (yet sufficient) set of components that have to be stopped. We plan to investigate how port connections could help provide an efficient solution to this problem.

Acknowledgements. The authors thank the anonymous reviewers for the usefulness and precision of their comments that allowed to greatly increase the quality of this paper.

REFERENCES

1. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
2. Frantisek Plásil, Dusan Balek, and Radovan Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. of the Int. Conf. on Configurable Distributed Systems*, pages 43–52, Washington, DC, USA, 1998. IEEE Computer Society.
3. E. Bruneton, T. Coupaye, and JB. Stefani. Fractal specification - v 2.0.3, February 2004. `http://fractal.objectweb.org/specification/index.html` [3 July 2008].
4. Bart George, Régis Fleurquin, and Salah Sadou. 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, July 2006.

5. Tomas Bures, Petr Hnetynka, and Frantisek Plásil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.
6. Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard. Automating the building of software component architectures. In Volker Gruhn and Flavio Oquendo, editors, *Software Architecture: 3rd European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA)*, volume 4344 of *LNCS*, pages 228–235. Springer, 2006.
7. Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.
8. Ivica Crnkovic. Component-based software engineering—new challenges in software development. *Software Focus, John Wiley & Sons*, 2(4):127–133, December 2001.
9. Remco M. Dijkman, Joao Paulo Andrade Almeida, and Dick A.C. Quartel. Verifying the correctness of component-based applications that support business processes. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proc. of the 6th Workshop on CBSE: Automated Reasoning and Prediction*, pages 43–48, Portland, Oregon, USA, May 2003.
10. Nenad Medvidovic and Richard N.Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.
11. Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Trans. Softw. Eng. Methodol.*, 9(3):239–272, 2000.
12. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. of the 8th European software engineering conference*, pages 109–120, New York, NY, USA, 2001. ACM Press.
13. Martin Mach, Frantisek Plásil, and Jan Kofron. Behavior protocols verification: Fighting state explosion. *International Journal of Computer and Information Science, ACIS*, 6(1):22–30, March 2005.
14. Viliam Holub and Frantisek Plásil. Reducing component systems' behavior specification. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 63–72, Washington, DC, USA, 2007. IEEE Computer Society.
15. Viliam Holub and Petr Tuma. Streaming state space: A method of distributed model verification. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 356–368, Shanghai, China, June 2007. IEEE Computer Society.
16. Kurt C. Wallnau. Volume III: A technology for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon University, Pittsburgh, OH, USA, April 2003.
17. Jiri Adamek and Frantisek Plásil. Partial bindings of components - any harm? In *APSEC '04: Proc. of the 11th Asia-Pacific Software Engineering Conference*, pages 632–639, Washington, DC, USA, 2004. IEEE Computer Society.
18. Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*, pages 287–325. Springer, 2003.
19. David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.
20. Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4):314–335, 1995.
21. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proc. of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.
22. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proc. of ICSE*, pages 187–197, Orlando, FL, USA, May 2002. ACM Press.
23. Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Softw. Eng.*, 21(4):373–386, 1995.
24. Robert John Allen. *A formal approach to software architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
25. Jasminka Matevska-Meyer, Wilhelm Hasselbring, and Ralf H. Reussner. A software architecture description supporting component deployment and system runtime reconfiguration. In *Proc. of the 9th Int. Workshop on Component-Oriented Programming (WCOP '04)*, Oslo, Norway, June 2004.
26. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of 20th Intl. Conf. on Software Engineering*, pages 177–187, Kyoto, Japan, April 1998. IEEE Computer Society.
27. P. Brada. Component change and version identification in SOFA. In *SOFSEM '99: Proc. of the 26th Conf. on Current Trends in Theory and Practice of Informatics*, pages 360–368, London, UK, 1999.

Springer-Verlag.

28. Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay. Automated and unanticipated flexible component substitution. In Helnz W. Schmidt, Ivica Crnkovic, Georges T. Heineman, and Judith A. Stafford, editors, *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE2007)*, volume 4608 of *LNCS*, pages 33–48, Medford, MA, USA, July 2007. Springer.
29. F. Plásil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.
30. OMG. Unified modeling language: Superstructure, version 2.0, 2002. `http://www.omg.org/uml`.
31. Ana Elisa Lobo, Paulo Asterio de C. Guerra, Fernando Castor Filho, and Cecilia Mary F. Rubira. A systematic approach for the evolution of reusable software components. In *ECOOP'2005 Workshop on Architecture-Centric Evolution*, Glasgow, july 2005. `http://wi.wu-wien.ac.at/home/uzdun/ACE2005/04-lobo.pdf` [4 July 2008].
32. Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Precalculating component interface compatibility using FCA. In Jean Diatta, Peter Eklund, and Michel Liquière, editors, *Proceedings of the 5th international conference on Concept Lattices and their Applications (CLA 2007)*, pages 241–252, Montpellier, France, October 2007.
33. C.B. Jones. *Systematic Software Development using VDM (2nd Edition).* Prentice-Hall, 1990.
34. J.-R. Abrial. *The B-Book, Assigning programs to meanings.* Cambridge University Press, 1996.
35. Rina Dechter. *Constraint processing.* Morgan Kaufmann, 2003.
36. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimisation, Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, 1982.
37. Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE '07)*, pages 342–357, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society.
38. Anthony J. Bagnall, Victor J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information & Software Technology*, 43(14):883–890, 2001.
39. Alejandra Cechich, Mario Piattini, and Antonio Vallecillo, editors. *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*. Springer, 2003.