International Journal of General Systems
Vol. 00, No. 00, October 2008, 1–23

Research paper

FCA-based service classification to dynamically build efficient software component directories

Gabriela Arévalo[a*], Nicolas Desnos [b], Marianne Huchard [c],

Christelle Urtado [b] and Sylvain Vauttier [b]

[a] *LIFIA - Facultad de Informática (UNLP) - La Plata - Argentina*;
[b] *LGI2P / Ecole des Mines d'Alès - Nîmes - France*;
[c] *LIRMM - CNRS and Univ. Montpellier - France*

(*released August 2008*)

Component directories index components by the services they offer thus enabling us to rapidly access them. Component directories are also the cornerstone of dynamic component assembly evolution when components fail or when new functionalities have to be added to meet new requirements. This work targets semi-automatic evolution processes. It states the theoretical basis of on-the-fly construction of component directories using Formal Concept Analysis based on the syntactic description of the services that components require or provide. In these directories, components are more clearly organized and new abstract and highly reusable component external descriptions suggested. Moreover, this organization speeds up both automatic component assembly and automatic component substitution.

Keywords: Component-Based Software Engineering, Component directories, Formal Concept Analysis, Component classification

1. Introduction

Component-based software engineering (CBSE) enables software applications to be built by assembling off-the-shelf components. To ease this process, components expose their external description: a component's set of required and provided interfaces corresponds to the syntactical description of the services the component provides to other components in its environment or requires from other components of its environment to execute itself. Previous work on automatic component assembly and dynamic component assembly evolution (Desnos *et al.* 2006, 2007, 2008) convinced us that an efficient component directory is needed. Indeed, searching in a directory for a component from a given repository that is compatible with, or substitutable for, a given component is a non-trivial task. Additionally, white-page-like directories, which represent the mostly used category of directories, are not suitable because they are not structured to enable the search for compatible or substitutable components.

The idea of this paper is to propose mechanisms to semi-automatically index software components through a yellow-page-like component directory that supports

*Corresponding author. Email: Gabriela.Arevalo@lifia.info.unlp.edu.ar

ISSN: 0308-1079 print/ISSN 1563-5104 online
© 2008 Taylor & Francis
DOI: 10.1080/0308107YYxxxxxxxx
http://www.informaworld.com

efficient search for components that are compatible or substitutable to a given component. Our approach relies on Formal Concept Analysis (FCA) that enables us to pre-calculate three categories of lattices:

- *Functionality signature lattices* order functionality signatures in a way that naturally eases their search and can be used for required and provided functionality connection or for required or provided functionality substitution. This category of lattices serves as the basis for building interface lattices.
- *Interface lattices* are more abstract than functionality signature lattices; they code information on functionality specialization that has been modeled in functionality signature lattices. They order component interfaces — organize service descriptions — in a way that naturally eases their search and can be used for required and provided interface connection or for required or provided interface substitution. This category of lattices serves as the basis for building component type lattices.
- *Component type lattices* are more abstract than interface lattices; they code the information on interface specialization that has been modeled in interface lattices. They order component types in a way that naturally eases their search and can be used for component connection or component substitution.

These lattices provide the architect or developer with intelligible classifications for functionality signatures, interfaces and component types. They enable us to separate the service compatibility calculus from the component search itself during the processes of assembly or component assembly evolution (component substitution).

Indeed, a component type lattice can be used as an index for the search of a compatible component (in order to build an assembly) or of a comparable component (in order to find a substitute). Furthermore, FCA creates new component external descriptions (new component types) that do not exist in the component repository but are more abstract and reusable than existing components. These new abstractions can be an opportunity for component developers to be guided during their engineering or re-engineering process. They can also enrich the repository.

The remainder of this paper is organized as follows. Section 2 shows an extension of object-oriented type theory to component types. Then, after recalling the basics of FCA and describing the example used in the paper, Section 3 shows how to build a lattice of functionality signatures and how to use it as a basis for component assembly or component substitution. Section 4 generalizes these results to entire interfaces and shows how to use the resulting interface lattice. Section 5 goes one step further in proposing a methodology to build and interpret a component lattice. To finish, Section 6 compares our approach to related existing work and Section 7 concludes and presents future research directions.

2. Functionality signatures and interface syntactical compatibility

This section explains how the syntactical compatibility of component interfaces can be calculated from functionality signatures which define the syntactical type of interfaces. The syntactical compatibility of interfaces is used to check the validity of connection and substitution operations on component assemblies. It statically asserts a certain level of coherence in a component assembly that, before semantic analysis or execution, provides early error detection and correction.

2.1 *Functionality signature compatibility in object-oriented programming*

In strongly-typed object-oriented programming languages (Cardelli 1984), method signature overriding is allowed in subclasses but constrained by rules that enforce

the substitutability of subclass instances towards superclass instances. Thus, a method signature in a subclass must have contravariant argument types and a covariant return type: argument types must be generalized and the return type must be specialized. Intuitively, a method implements a service provided by an object: when the method is called, assuming that sufficient information is received (as specified by argument types), a result of the defined return type is sent back. This corresponds to the concept of software contract, introduced by Meyer (1991) to reason about interactions between objects. Following the above rules, an instance of a class can replace an instance of one of its superclasses because it provides at least the same services, but is allowed to require less invocation information and to return a richer result.

These principles are also used to define relaxed matching schemes used to retrieve a class or a functionality from a repository (Zaremski and Wing 1995). A request is expressed as the signature of the functionality that is searched for. Any functionality the signature of which specializes (overrides) the requested signature is returned as an approximate but still (type-) compatible answer.

2.2 *Functionality signatures and component interface specification*

An interface is a type that collects functionality signatures; it is used to qualify the collaborations a component can establish with other components. An interface is also a communication point through which a component exchanges service request and response messages with another component. Messages are sent and received along connections linking the interfaces of a component to compatible interfaces of other components (Szypersky *et al.* 2002). Comparing the syntactical types of two interfaces amounts to compare pairs of functionality signatures from both interfaces (Zaremski and Wing 1997). But in contrast with object models, a direction is added to the definition of interfaces in order to specify whether a component is a client (*i.e.,* uses the interface to require a service) or a server (*i.e.,* uses the interface to provide a service). Thus, two kinds of compatibilities can be verified between interfaces: a connection compatibility between a client interface and a server interface or a substitution compatibility between interfaces that have the same direction. The connection or substitution compatibility of two components can in turn be determined by verifying the connection or substitution compatibility of pairs of interfaces from both components.

In this paper, functionality signatures are defined by a name, a list of argument types and a return type. As in classical programming languages, names are used as the primary semantic element to match functionalities. Then, the types of the IN-parameters and OUT-parameters of homonymic functionalities are considered. For the sake of simplicity, only a single OUT-parameter (the functionality result) is used in this paper. But the same principles can be applied to any OUT-parameter when multiple OUT-parameters are used in a functionality signature.

Figure 1 shows an example of different signatures for homonymic functionalities named `create`, associated with both required and provided component interfaces. The data type hierarchy used to define parameter types is presented in Figure 1(d). The different cases of functionality signature specialization are illustrated: argument type specialization (*cf.* Figure 1(a)), result type specialization (*cf.* Figure 1(b)) and argument addition into the IN-parameter set (*cf.* Figure 1(c)).

When associated with a provided interface, a functionality signature has the same semantics as in object-oriented programming: the argument types define what the server component requires to receive in order to execute its service and the return type defines what result it commits to provide. When associated with a required

interface, a functionality signature specifies the service that is searched for by a client component: the argument types define the invocation information that the client component will send to a server component and the return type defines the type of the result it requires.

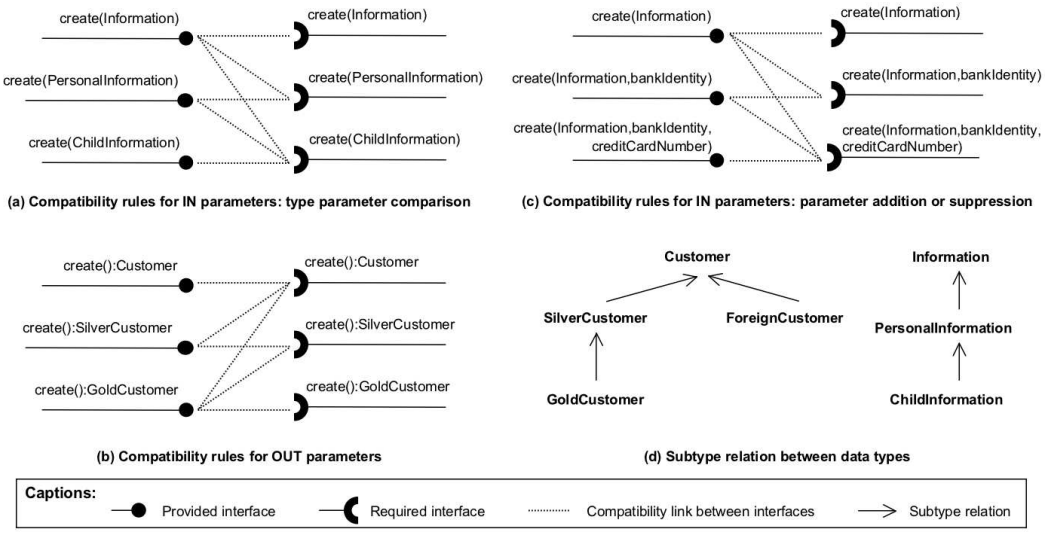

Figure 1. Interface compatibility when types and number of parameters vary.

2.3 *Functionality signature specialization and provided interface substitution*

Zaremski and Wing (1997) present functionality signature matching based on pre- and post- conditions. Consider a provided interface I_1, which holds a functionality of signature $S = f(X\ x) : Z$. As informally stated above, its software contract corresponds to the following pre-condition and post-condition:

$$S_{pre}(x) : Type(x) \leq X$$
$$S_{post}(x) : Type(f(x)) \leq Z$$

Let us consider another provided interface I_2, which holds a functionality of signature $T = f(L\ l) : M$, along with its pre-condition and post-condition:

$$T_{pre}(l) : Type(l) \leq L$$
$$T_{post}(l) : Type(f(l)) \leq M$$

To soundly substitute I_2 to I_1 in an assembly, the following predicate must hold:

$$Substitution_{provided}(I_2, I_1) = S_{pre}(x) \Rightarrow T_{pre}(x) \wedge T_{post}(x) \Rightarrow S_{post}(x)$$

This verifies that f in I_2 can execute the same invocations as f in I_1; second, it verifies that the results returned by f in I_2 can be used instead of the results returned by f in I_1.

To be true, the predicate entails that:

$$X \leq L \text{ (indeed, } Type(x) \leq X \wedge X \leq L \Rightarrow Type(x) \leq L),$$
$$M \leq Z \text{ (indeed, } Type(f(x)) \leq M \wedge M \leq Z \Rightarrow Type(f(x)) \leq Z).$$

This respectively corresponds to a contravariant specialization of the argument types and to a covariant specialization of the result type between the two functionality signatures, as previously presented for object-oriented languages. A provided

interface can be replaced by another provided interface with more specific functionality signatures, following the above specialization rules.

For example (*cf.* Figure 1(a)), a provided interface holding the `create(Information)` signature can be substituted to a provided interface holding the `create(PersonnalInformation)` signature (contravariant specialization of the argument type). Similarly (*cf.* Figure 1(b)), a provided interface holding the `create():GoldCustomer` signature can be substituted to a provided interface holding the `create():SilverCustomer` signature (covariant specialization of the result type).

2.4 *Functionality signature specialization and required interface substitution*

Let us now consider a required interface I_3, which holds a functionality of signature $S = f(X\ x) : Z$. The pre-condition and post-condition corresponding to its software contract are the same as for a provided interface but, as discussed above, their semantics are converse. Indeed, x now represents the data the client component commits to send and $f(x)$ the data the client expects to receive:

$$S_{pre}(x) : Type(x) \leq X$$
$$S_{post}(x) : Type(f(x)) \leq Z$$

Let us also consider another required interface I_4, which contains a functionality of signature $T = f(L\ l) : M$. This corresponds to the same pre-condition and post-condition as above:

$$T_{pre}(l) : Type(l) \leq L$$
$$T_{post}(l) : Type(f(l)) \leq M$$

To soundly substitute I_4 to I_3 in an assembly, the following predicate must hold:

$$Substitution_{required}(I_4, I_3) = T_{pre}(x) \Rightarrow S_{pre}(x) \wedge S_{post}(x) \Rightarrow T_{post}(x)$$

This firstly verifies that the client component holding I_4 will call f in the same way as the client component holding I_3 (to have the guarantee that the connected server component can execute all invocations); secondly, this verifies that the results received by I_3 will also satisfy the requirements of the client component holding I_4.

To be true, the predicate entails that:

$$L \leq X \text{ (indeed, } Type(x) \leq L \wedge L \leq X \Rightarrow Type(x) \leq X),$$
$$Z \leq M \text{ (indeed, } Type(f(x)) \leq Z \wedge Z \leq M \Rightarrow Type(f(x)) \leq M).$$

This respectively corresponds to a covariant specialization of the argument types and a contravariant specialization of the result type between the two functionality signatures. Unsurprisingly, the specialization rules for functionality signatures in required interfaces are the opposite of those which apply to provided interfaces. Here again, following the above rules, a required interface can be replaced by another required interface with more specific functionality signatures.

For example (*cf.* Figure 1(a)), a required interface holding the `create(ChildInformation)` signature can be substituted to a required interface holding the `create(PersonnalInformation)` signature (covariant specialization of the argument type). Similarly (*cf.* Figure 1(b)), a required interface holding the `create():Customer` signature can be substituted to a required interface holding the `create():SilverCustomer` signature (contravariant specialization of the result type).

2.5 *Functionality signature specialization and interface connection*

Finally, let us again consider the provided interface I_1 and the required interface I_4. To soundly connect I_1 to I_4, the following predicate must hold:

$$Connection(I_4, I_1) = T_{pre}(x) \Rightarrow S_{pre}(x) \wedge S_{post}(x) \Rightarrow T_{post}(x)$$

This firstly verifies that any data sent by the client component holding I_4 can effectively be used by the server component holding I_1 to execute f; secondly, this verifies that the data sent by the server component holding I_1 corresponds to the result expected by the client component holding I_4.

To be true, the predicate entails that:

$$L \leq X \text{ (indeed, } Type(x) \leq L \wedge L \leq X \Rightarrow Type(x) \leq X),$$
$$Z \leq M \text{ (indeed, } Type(f(x)) \leq Z \wedge Z \leq M \Rightarrow Type(f(x)) \leq M).$$

This corresponds to a contravariant specialization of argument types and a covariant specialization of the result type between the two functionality signatures. The functionality signatures associated with a required interface of the client component must be more generic than the functionality signature associated with the provided interface of the server component.

For example (*cf.* Figure 1(a)), a required interface holding the `create(PersonalInformation)` signature can be connected to a provided interface holding the `create(Information)` signature (contravariant specialization of the argument type). Similarly (*cf.* Figure 1(b)), a required interface holding the `create():Customer` signature can be connected to a provided interface holding the `create():SilverCustomer` signature (covariant specialization of the result type).

2.6 *Functionality signature specialization and parameter addition or suppression*

A special case of parameter type generalization is now considered. When a parameter type is generalized in a functionality signature, it conceptually means that the specification becomes less demanding on parameters. The objects of the `Object` type (root of the object type hierarchy) are the objects which contain the least data. We extend the generalization principle by stating that `void` is the root type in our system and that it further generalizes the `Object` type.

This way, a special case of parameter type generalization is to set a parameter type to `void`. Any data, including no data, becomes suitable for this parameter. As this parameter is optional, it is possible to remove the parameter from the functionality signature. We therefore consider suppressing a parameter as a special case of parameter type generalization.

Conversely, it is possible to add an extra parameter of type `void` to a functionality signature without changing its semantics (this additional parameter can always be ignored). The type of such a parameter can then be specialized in the process of functionality signature specialization, thus becoming a parameter of a concrete type. We therefore consider parameter addition as a special case of parameter type specialization.

For example (*cf.* Figure 1(c)), a provided interface holding the `create(Information)` signature can be substituted to a provided interface holding the `create(Information,BankIdentity)` signature, as the former signature is obtained by removing the second parameter of the latter signature (contravariant specialization of the parameter type). Similarly, a required interface holding the `create(In-`

`formation,BankIdentity)` signature can be substituted to a required interface holding the `create(Information)` signature, as the former signature is obtained by adding a second parameter to the latter signature (covariant specialization of a virtual second parameter of type `void`).

2.7 *Discussion*

In Zaremski and Wing (1997), which proposes an extensive study and classification of functionality signature matching, the above predicates correspond to a kind of functionality signature matching called "plug-in" matching. It is used to verify that the code of a functionality can be plugged into some other code, to handle some expected behavior, as specified by a syntactical signature. We have adapted this generic functionality signature matching principle to the specific concepts of component models, namely the syntactical coherence of interface connection and substitution.

Our formalization shows that checking the coherence of these operations amounts to verifying the existence of specialization relations between functionality signatures. Thus, we studied how to build specialization hierarchies of functionality signatures, interfaces and component types. We intend to use these hierarchies as a practical, systematic and efficient means to set up and structure a component directory, where components are indexed by the type of services they provide and require, in other words, a trading service for component-based platforms (Iribarne *et al.* 2004)).

The next sections describe how an FCA-based approach to this problem can be used to build the necessary specialization lattices. It is to be noticed that, at any step, a single lattice is sufficient to compare both required and provided elements for both substitution and connection. Indeed, as shown previously, only two specialization rules are used, which are converse.

3. Lattice of functionality signatures

The substitutability rules presented in the previous section can be considered as the basis of a specialization relationship among functionalities: a functionality that can substitute for another can be considered as its specialization. Existing functionalities can thus be organized — classified — in a hierarchy based on their substitutability relationships. Furthermore, this section will show that FCA provides a finer-grained classification. After recalling the basics of Formal Concept Analysis, we show how it can be used to build a lattice of functionality signatures and how the lattice can then be interpreted and used.

3.1 *A survival kit for Formal Concept Analysis*

The classification we build is based on the partially ordered structure known as *Galois connection-based lattice* (Birkhoff 1940, Davey and Priestley 1991) or *concept lattice* (Wille 1982) which is induced by a context K, composed of a binary relation R over a pair of sets O (*objects*) and A (*attributes*) (Table 1). A formal concept C is a pair of corresponding sets (E, I) such that:

$$E = \{ \text{e} \in O | \ \forall \text{ i} \in \text{I}, (\text{e}, \text{i}) \in R\} \quad \text{is called } \textit{extent} \text{ (covered objects)},$$
$$I = \{ \text{i} \in A | \ \forall \text{ e} \in \text{E}, (\text{e}, \text{i}) \in R\} \quad \text{is called } \textit{intent} \text{ (shared features)}.$$

For example, $(\{1,2\},\{b,c\})$ is a formal concept because objects 1 and 2 exactly share attributes b and c (and vice-versa). On the contrary, $(\{2\},\{b,c\})$ is not a formal concept.

Furthermore, the set of all formal concepts $\mathcal{C}$ constitutes a lattice $\mathcal{L}$ when provided with the following specialization order based on intent / extent inclusion:

$$(E_1, I_1) \leq_{\mathcal{L}} (E_2, I_2) \Leftrightarrow E_1 \subseteq E_2 \text{ (or equivalently } I_2 \subseteq I_1).$$

Figure 3.1 shows the Hasse diagram of $\leq_{\mathcal{L}}$.

Table 1. Binary relation of $K = (O, A, R)$ where O=$\{1, 2, 3, 4, 5, 6\}$ and A= $\{a, b, c, d, e, f, g, h\}$.

	a	b	c	d	e	f	g	h
1		×	×	×	×			
2	×	×	×				×	×
3	×	×				×	×	×
4				×	×			
5			×	×				
6	×							×

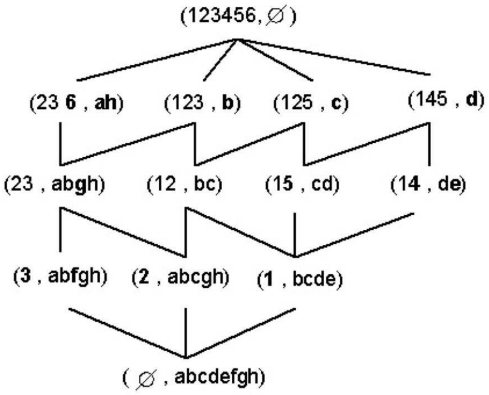

Figure 2. Hasse diagram of the concept lattice $\mathcal{L}$.

3.2 *Example of an online bookstore application*

In the rest of this article, we will use, as an illustration, the example of an online bookstore application that targets both the adult and children audiences (*cf.* Figure 3(a) to see the hierarchy of product types). Two categories of customers can interact with this application. Adults can save favorite book lists (as wish lists) through the application or shop for books following various protocols defined according to a client typology (*cf.* Figure 1(d)). Children can establish children book wish lists that constitute virtual orders that adults can offer them as soon as their parents obtain the `SilverCustomer` client category. For this online bookstore application, we have a component repository (*cf.* Figure 3(b)) in which we can see various components to manage orders (by adults or children) and various components to manage customer lists. These components each expose an interface list the types of which are enumerated in Figure 3(c).

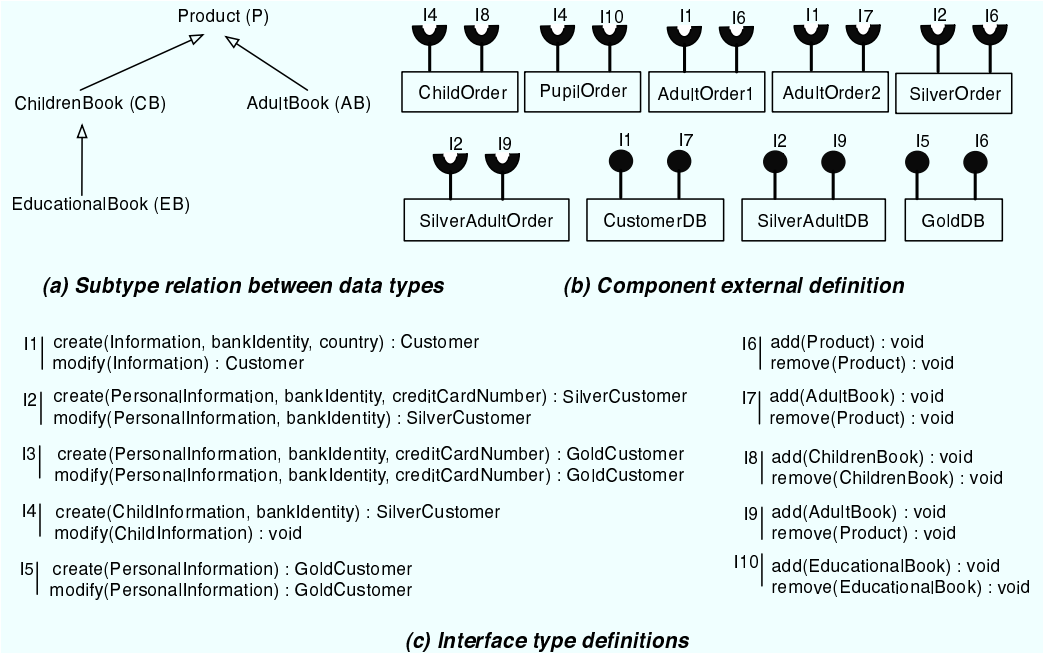

Figure 3. Data types, interfaces and components of an online bookstore application.

3.3 *Building the functionality signature lattice*

We explain here the construction of the required functionality signature lattice. As provided functionality signatures are reversely ordered, the lattice we obtain can also be used to deal with them, when considered upside down.

We illustrate our explanation considering the required functionality `create(PI, BI,CCN):SC` as it is described by Table 2. At first, for each `create` functionality whose signature is held by one of the interfaces of Figure 3, attributes are deduced from IN and OUT parameter types that explicitly appear in the signature. These attributes are marked using the $\times$ symbol in Table 2: `create(PI,BI,CCN):SC` is thus described explicitly by attributes `IN:PI`, `IN:BI`, `IN:CCN` and `OUT:SC`. Then, we infer attributes (marked with a $\otimes$ symbol in Table 2) when their types are compatible, regarding specialization of signatures. Here are our inference rules:

- IN parameters. As explained previously, if a required functionality sends a parameter of some type, it implicitly sends a parameter of any more general type. For example, the `IN:I` attribute is inferred when the `IN:PI` attribute is already present.
- OUT parameters. If a required functionality expects to receive a return value of a type, any return value of a more specific type is also suitable. For example, the `OUT:GC` attribute is inferred when the `OUT:SC` attribute is already present.

Figure 4 depicts the concept lattice corresponding to the binary relation shown in Figure 2, built with the GaLicia FCA tool (GaLicia 2002). Concepts are presented using reduced intents and extents (resp. denoted by *ReducedI* et *ReducedE*) for readability sake: an object (signature) that belongs to the reduced extent of a concept is inherited by all concepts that are above (down-to-up inheritance); similarly, a property (IN or OUT parameter type) that belongs to the reduced intent of a concept is inherited by all concepts that are below (up-to-down inheritance).

Table 2. R_{create} context describing signatures of the required `create` functionality through its parameters.

	IN parameters						OUT param.		
	I	PI	CI	BI	CCN	Co	C	SC	GC
create(I,BI,Co):C	×			×		×	×	⊗	⊗
create(PI,BI,CCN):SC	⊗	×		×	×			×	⊗
create(PI,BI,CCN):GC	⊗	×		×	×				×
create(CI,BI):SC	⊗	⊗	×	×				×	⊗
create(PI):GC	⊗	×							×

I	Information
PI	PersonalInfo.
CI	ChildInfo.
BI	BankIdentity.
CCN	CreditCardNb
Co	Country
C	Customer
SC	SilverCustomer
GC	GoldCustomer
FC	ForeignCustomer

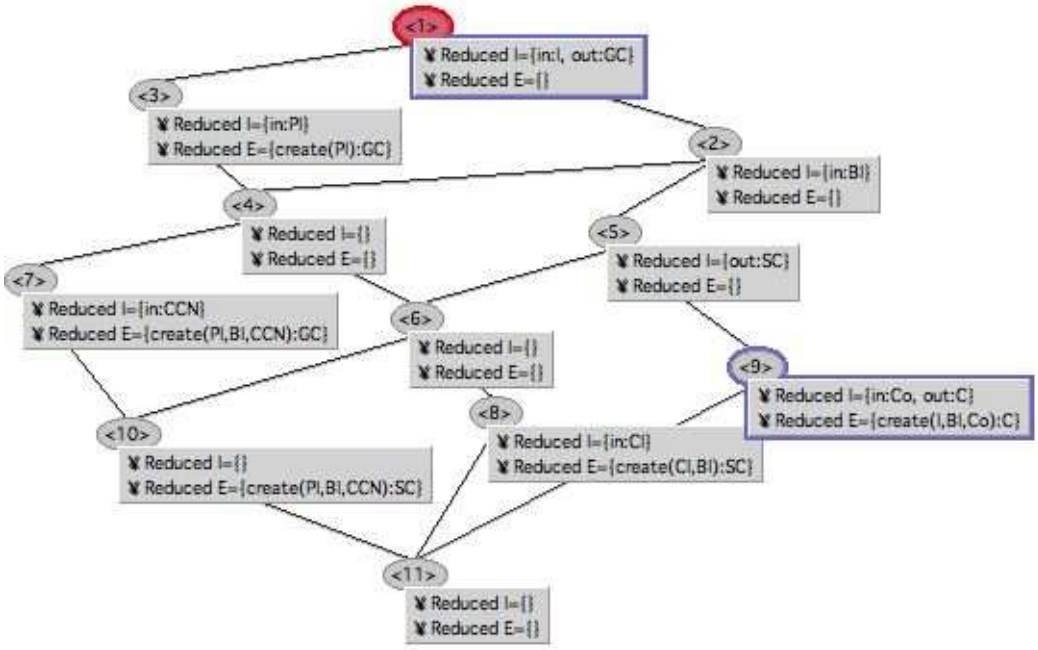

Figure 4. Signature lattice $\mathcal{L}_{create}$ for the `create` functionalities.

3.4 *Using the functionality signature lattice*

The functionality signature lattice can be used in various types of situations related to component connection or substitution.

Let us consider the lattice of Figure 4 with the viewpoint of required functionalities. In this lattice, `create(PI):GC` is represented by concept C_3 while `create(CI,BI):SC` is represented by concept C_8. Concept C_3 is more general than concept C_8 which can be interpreted as: concept C_8 can replace concept C_3. In a component assembly, a connection to a required functionality corresponding to concept C_3 can be replaced by a connection to a required functionality corresponding to concept C_8. In the general case, when there is a path between two concepts, the more specific (which has more properties) can replace the more general (which has a subset of properties) when the more general concept is connected (*cf.* Figure 5(a)). The same lattice can also be used to substitute a provided functionality when read upside down (*cf.* Figure 5(b)). This generalizes as follows.

Property 3.1 Functionality substitution. Let C_{father}, C_{son} be two concepts of the signature lattice of functionality f, such that $C_{son} \leq_{\mathcal{L}_f} C_{father}$. Functionalities of C_{son} can replace functionalities of C_{father} when the functionalities are required. Opposite replacement applies when the functionalities are provided.

Both provided and required points of view can be combined to address com-

ponent connection. Let us consider the `create(PI,BI,CCN):GC` signature (concept C_7). The corresponding required functionality can obviously connect to the provided functionality that has the same signature (`create(PI,BI,CCN):GC`). Given the substitution rule, provided functionalities which are upper in the lattice, such as provided `create(PI):GC` (concept C_3), can be connected to required `create(PI,BI,CCN):GC` (*cf.* Figure 5(c)). Using the same rule in the symmetric way, required functionalities which are below in the lattice, such as required `create(PI,BI,CCN):SC` (concept C_{10}), can be connected to provided `create(PI,BI,CCN):GC`. By transitivity, we can deduce that required `create(PI, BI,CCN):SC` can be connected to provided `create(PI):GC`. This is expressed in the following connection rule that formalizes how valid functionality connection can be deduced from the lattice.

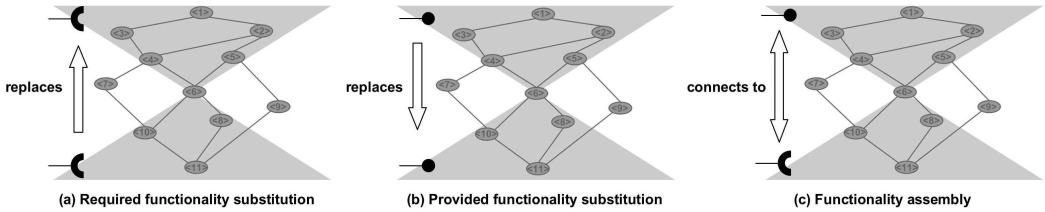

Figure 5. Interpretation of the lattice of functionality signatures.

Property 3.2 Functionality connection rule. Let C, C_{father}, C_{son} be three concepts of the signature lattice of functionality f such that $C_{son} \leq_{\mathcal{L}_f} C \leq_{\mathcal{L}_f} C_{father}$, required functionalities of C_{son} can be connected to provided functionalities of C_{father}.

4. Interface lattice

Components are reusable software entities that are chosen off-the-shelf and fulfill high-level goals (database component, planning component, and so on). Interfaces play an important role to achieve these goals by grouping functionalities that have close semantics and may participate together in potential collaborations. Component assembly is based mainly on the connection of compatible interfaces in a higher abstraction level than simple functionalities.

Considering included functionalities, the interfaces can be provided with a specialization order in a natural way. This "natural" classification simply uses the inclusion relation between sets of functionalities in the interfaces and can equally benefit from FCA to look for factorizable functionalities (in our case `remove(P)` can be factored out).

Then, if we consider substitution or connection, we can improve our search and discover more pertinent abstractions when using the abstractions discovered in the functionality signature lattice. Lattices of the `modify`, `add` and `remove` functionalities of our example are built similarly to the lattice of the `create` functionality. Tables 3 and 4 detail the contexts, while Figure 6 and 7 show the corresponding lattices. As we have observed, these abstractions on the signatures are the concepts the extent of which has a set of signatures (the signatures covered by the concept) and the intent of which has a set of attributes describing the signature (IN and OUT parameters). For each concept, we can calculate a corresponding canonical signature. We show an example before giving the general definition.

Figure 4 shows the concepts built using the binary relation described in Table 2. A concept the reduced extent of which has an original signature (*e.g.,* concept

Table 3. Context R_{modify} describing the signatures of the required `modify` functionalities.

	In parameters					Out param.			
	I	PI	CI	BI	CCN	void	C	SC	GC
modify(I):C	×						×	⊗	⊗
modify(PI,BI):SC	⊗	×		×				×	⊗
modify(PI,BI,CCN):GC	⊗	×		×	×				×
modify(CI):void	⊗	⊗	×			×	⊗	⊗	⊗
modify(PI):GC	⊗	×							×

Table 4. Context R_{add} describing the signatures of the required `add` functionalities. The context R_{remove} is identical.

	In parameters				Out param.
	P	AB	CB	EB	void
add(P):void	×				×
add(AB):void	⊗	×			×
add(CB):void	⊗		×		×
add(EB):void	⊗		⊗	×	×

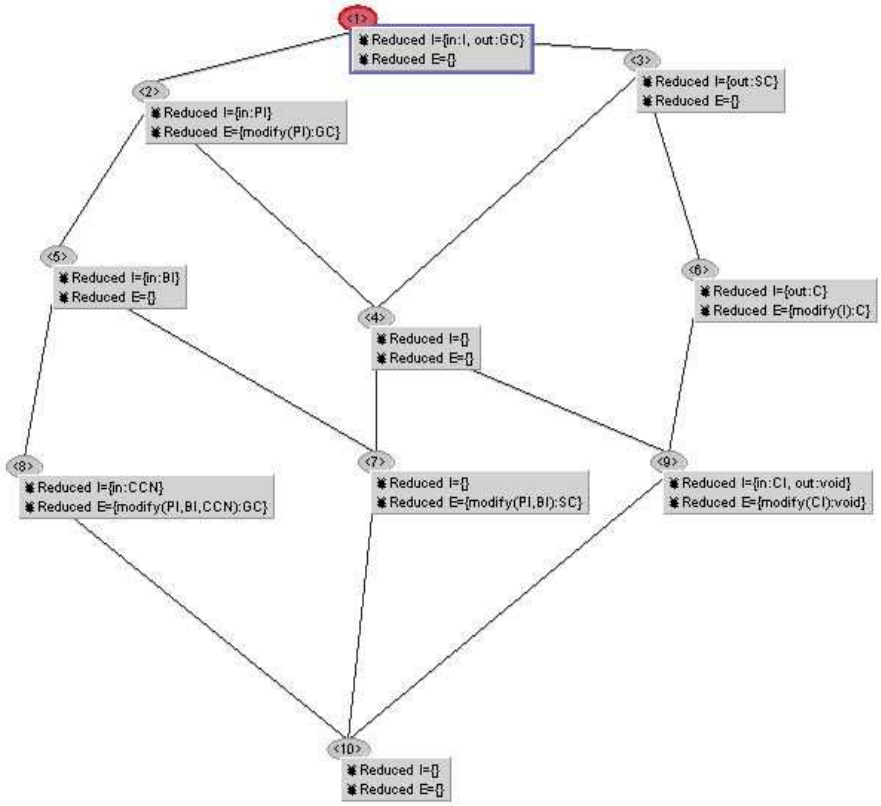

Figure 6. Signature lattice $\mathcal{L}_{modify}$ for the `modify` functionalities

C_9) exactly represents that signature (*e.g.,* `create(I,BI,Co):C`). A concept the reduced extent of which is empty can be interpreted as a new signature that we can infer starting from the attributes inherited by the concept, and considering only the more specific ones. For example, concept C_6 of Figure 4 inherits attributes `in:I`, `in:PI`, `in:BI`, `out:GC`, `out:SC`. In case of required signatures, `in:PI` is more specific than `in:I` meanwhile `out:SC` is more specific than `out:GC`. Concept C_6 can be then interpreted as signature `create(PI,BI):SC` which we call the canonical signature of the concept. This enables us to build an interface description based on the set of original signatures completed by all the signatures created in the generalization process (*cf.* Table 5).

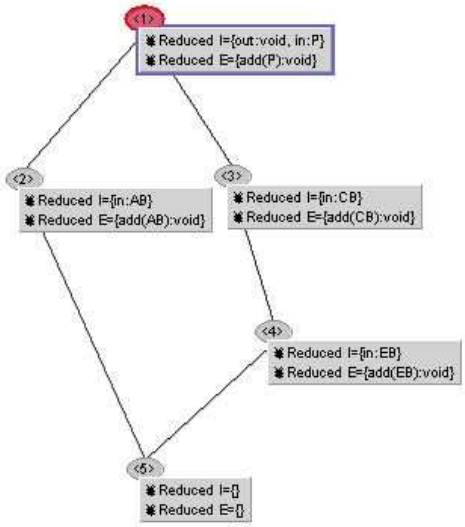

Figure 7. Signature lattice $\mathcal{L}_{add}$ for the `add` functionalities. The lattice $\mathcal{L}_{remove}$, isomorphic to $\mathcal{L}_{add}$, is not represented.

Definition 4.1 Canonical functionality signature of a concept**:** Let C be a concept in a signature lattice $\mathcal{L}_f$ which describes functionality f and $\leq_{Types}$, the specialization partial order on parameter types. $\sigma(C)$, the canonical signature of C, is defined as follows:

- If $ReducedE(C) = \{s\}$, then $\sigma(C) = s$.
- If $ReducedE(C) = \varnothing$, then $\sigma(C) = f(i) : o$, where $i = min_{\leq_{Types}}\{T|\text{IN} : T \in Intent(C)\}$ and where $o = max_{\leq_{Types}}\{T|\text{OUT} : T \in Intent(C)\}$.

This exact description enables us to build more pertinent interface generalizations than those we obtained with the "natural" classification of interfaces. It is used as follows to build interface descriptions within the new context $R_{IntSigCar}$.

- The canonical signatures are used as attributes in the formal context.
- When an interface I has a signature s in a functionality f in its original description, if we denote by C the concept such that $\sigma(C) = s$, we associate to the interface the attribute s and all the canonical signatures of the concepts that are upper of C in the lattice:
$R_{IntSigCar} = \{(I, sc)|s \textit{ belongs to the definition of } I, sc = \sigma(C_{father}),$
$C_{father} \geq_{\mathcal{L}_f} C \textit{ with } s = \sigma(C)\}$.

For example, interface `I1` holds the signature `create(I,BI,Co):C`. This signature is the canonical signature of concept C_9 in lattice $\mathcal{L}_{create}$. In Tab. 5, we associate `I1` to `create(I,BI,Co):C` (marked with symbol $\times$) and we equally associate to `I1` the canonical signatures of all concepts of $\mathcal{L}_{create}$ that are upper of C_9. That results in the following signatures (marked with symbol $\otimes$) : `create(I,BI):SC` (concept C_5), `create(I,BI):GC` (concept C_2), and `create(I):GC` (concept C_1). From required functionality viewpoint, these signatures are generalizations of the original signature `create(I,BI,Co):C` (with the semantics of substitutability).

The built lattice $\mathcal{L}_I$ (*cf.* Figure 8) shows specialization relations between interfaces. These relations show possible connections or substitutions which are deduced from the previously mentioned rules on functionality signatures that are extended to interfaces (repeatedly applied to all signatures that constitute these interfaces).

For example, the required interface `I10` can be connected to provided interface

Table 5. Context $R_{IntSigCar}$ encoding required interfaces using signature generalizations. Rows: interfaces. Columns: canonical signatures and concepts.

	create											modify									
	create(I):GC — C_1	create(I,BI):GC — C_2	create(PI):GC — C_3	create(PI,BI):GC — C_4	create(I,BI):SC — C_5	create(PI,BI):SC — C_6	create(PI,BI,CCN):GC — C_7	create(CI,BI):SC — C_8	create(I,BI,Co):C — C_9	create(PI,BI,CCN):SC — C_{10}	create(CI,BI,CCN,Co):C — C_{11}	modify(I):GC — C_1	modify(PI):GC — C_2	modify(I):SC — C_3	modify(PI):SC — C_4	modify(PI,BI):GC — C_5	modify(I):C — C_6	modify(PI,BI):SC — C_7	modify(PI,BI,CCN):GC — C_8	modify(CI):void — C_9	modify(CI,BI,CCN):void — C_{10}
I1	⊗	⊗			⊗				×			⊗		⊗			×				
I2	⊗	⊗	⊗	⊗	⊗	⊗	⊗			×		⊗	⊗	⊗	⊗	⊗		×			
I3	⊗	⊗	⊗	⊗			×					⊗	⊗			⊗			×		
I4	⊗	⊗	⊗	⊗	⊗	⊗		×				⊗	⊗	⊗	⊗		⊗			×	
I5	⊗		×									⊗	×								
I6																					
I7																					
I8																					
I9																					
I10																					

	add					remove				
	add(P):void — C_1	add(AB):void — C_2	add(CB):void — C_3	add(EB):void — C_4	add(AB,EB):void — C_5	remove(P):void — C_1	remove(AB):void — C_2	remove(CB):void — C_3	remove(EB):void — C_4	remove(AB,EB):void — C_5
I1										
I2										
I3										
I4										
I5										
I6	×					×				
I7	⊗	×				×				
I8	⊗		×			⊗		×		
I9	⊗	×				×				
I10	⊗		⊗	×		⊗		⊗	×	

`I6`. Still, required interface `I10` (C_{10}) can replace required interface `I6` (C_2) . We see that a manual or automatic search of components is faster with this lattice that defines a search index. We thus avoid looking at all components in the repository since we only look for relevant branches. Let us imagine the case in our example where component `SilverAdultOrder` searches, logically, to be connected to component `SilverAdultDB` usually present in the system that is temporarily unavailable. The relation in the lattice, starting from the expected required interface I_9 (C_5), enables us to immediately find (just traversing the edge that goes from concept C_5 to concept C_2, that possesses the I_6 interface) that component `GoldDB` could be used as a replacement. Temporarily the user will benefit of a higher service in replacement of a missing service.

In the lattice, we also find new interfaces, obtained using the existing interface generalization. Starting from functionalities discovered in the first lattice, the technique can then infer a new interface, including at least this shared functionality. Here we see one of the main advantages of FCA-based techniques compared to simple calculation of signature comparison: new signatures appear, and thus we have new interfaces more abstract than existing ones. The following generalization step is to use this lattice to build a component lattice. This latter lattice is more interesting for designers who can be guided when creating more general new components, as well as for assemblers who can consult an organized library rather than

just a flat set of artifacts.

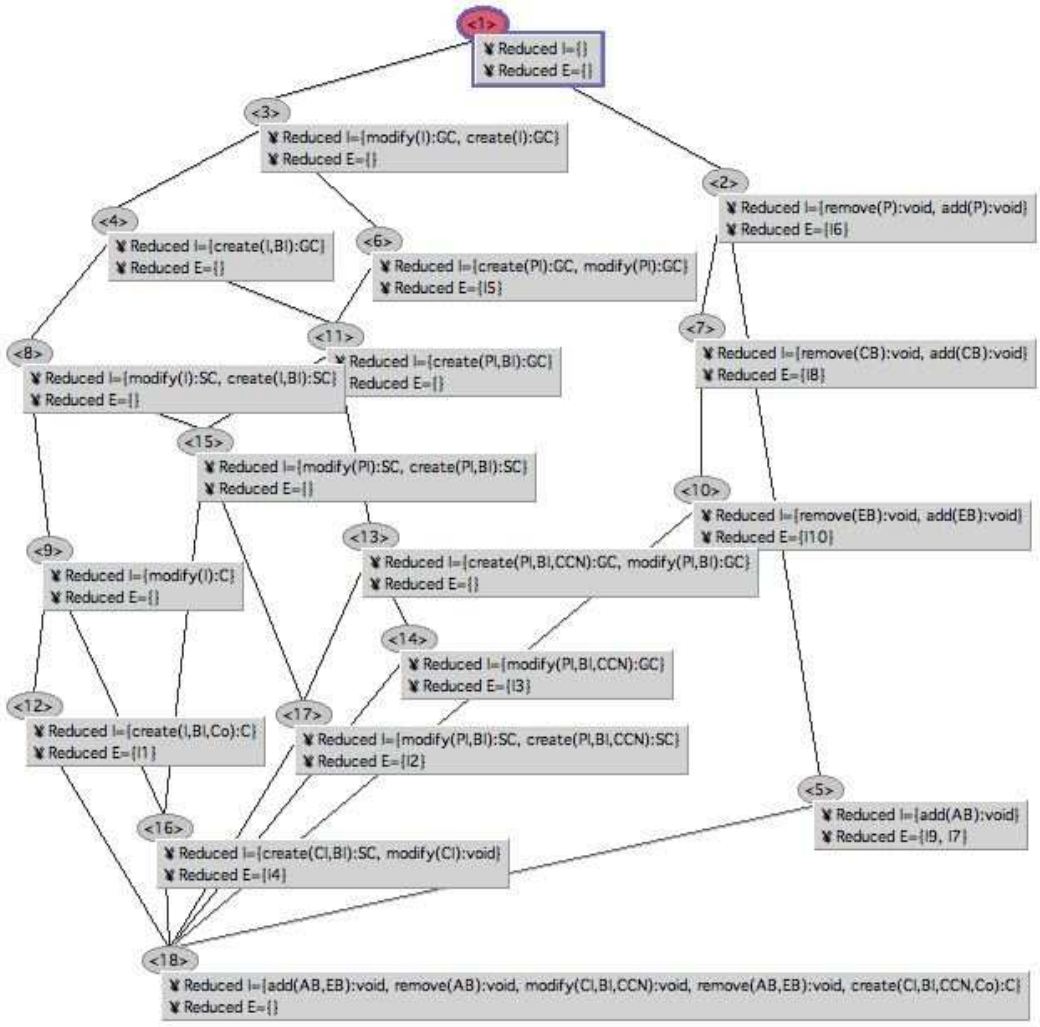

Figure 8. Interface lattice $\mathcal{L}_I$ using the functionality signature lattice.

5. Lattice of component types

In this section, we first propose a solution to build the lattice of component types. The technique used to do so is the same as the one previously used for interfaces: the interface lattice helps enrich the description of the formal context that will be used to build the component type lattice. Then, the remainder of the section shows possible uses of this lattice.

5.1 *Definition of the lattice of component types*

Component types are described by their required and provided interfaces. This information can be organized by specialization, but, similarly to that done with interfaces, component types can benefit from both the specialization relationships between interfaces and the discovered interfaces obtained from the interface lattice. We thus get an enrichment of the description of components and a more precise classification, offering more abstractions.

The first phase of the building process entailed the introduction of the notion of a "canonical interface" associated to an interface concept. This notion is similar to the canonical functionality signature corresponding to a signature concept that we defined above. Let us just mention that our analysis is still based on the case of required interfaces.

Definition 5.1 Required canonical interface corresponding to an interface concept: Let C be a concept in the interface lattice $\mathcal{L}_I$. The corresponding canonical interface $I(C)$ is defined as follows:

- If $ReducedE(C) \supseteq \{I\}$, then $I(C) = I$. We can choose any interface in the reduced extent because they are all equivalent.
- If $ReducedE(C) = \varnothing$, then $I(C) = min_{\leq_{SigCar}}\{s \in Intent(C)\}$. The canonical interface gathers more specialized signatures from the set of canonical signatures that forms the intent. The order $\leq_{SigCar}$ between canonical signatures is naturally inferred from the specialization relationship between concepts of the lattice $\mathcal{L}_f$: $s_{son} \leq_{SigCar} s_{father}$ iff $s_{son} = \sigma(c_{son})$, $s_{father} = \sigma(c_{father})$ and $c_{son} \leq_{\mathcal{L}_f} c_{father}$.

Canonical interfaces found in the lattice are all the original interfaces (`I1` to `I6`, `I8` and `I10`, and a single interface corresponding to the {`I7`,`I9`} interface pair) to which new abstract interfaces are added by the classification process. These new abstract interfaces are described by their signature set (*cf.* Tab. 6).

Table 6. New canonical interfaces.

Int. name	Signature set	Concept
I11	{}	C_1
I12	$\{create(I) : GC; modify(I) : GC\}$	C_3
I13	$\{create(I, BI) : GC; modify(I) : GC\}$	C_4
I14	$\{create(I, BI) : SC; modify(I) : SC\}$	C_8
I15	$\{create(I, BI) : SC; modify(I) : C\}$	C_9
I16	$\{create(PI, BI) : GC; modify(PI) : GC\}$	C_{11}
I17	$\{create(PI, BI) : SC; modify(PI) : SC\}$	C_{15}
I18	$\{create(PI, BI, CCN) : GC; modify(PI, BI) : GC\}$	C_{13}
I19	$\{create(CI, BI, CCN, Co) : C; modify(CI, BI, CCN) : void;$ $add(AB, EB) : void; remove(AB, EB) : void\}$	C_{18}

We then set up a relation $R_{CompCanInt}$ between component types and canonical interfaces including their orientation (required or provided) (*cf.* Tab. 7). The rows represent components, the columns interfaces. Interface identification (in column heads) combines the two interface orientations (noted `req:` and `pro:`) with each canonical interface name and is followed by their concept number in the interface lattice. For example, column 1 corresponds to the canonical required interface `I1`, associated to concept C_{12} (as `I1` is member of its reduced extent). Column 11 corresponds to the canonical required interface `I12`, associated to concept C_3.

Definition 5.2 Component relation $R_{CompCanInt}$:
Component types are the formal objects while canonical interfaces are the formal attributes. Let C be a component and I an interface, $(C, I) \in R_{CompCanInt}$ iff one of the following properties is true:

- I is declared by C,
- $I \geq_{\mathcal{L}_I} J$ and J is declared by C.

Figure 9 shows lattice $\mathcal{L}_C$ of component types. The following section will show how it can be used.

Table 7. The component context $R_{CompCanInt}$.

	req:I1 - c_{12}	req:I2 - c_{17}	req:I3 - c_{14}	req:I4 - c_{16}	req:I5 - C_6	req:I6 - C_2	req:I7,I9 - C_5	req:I8 - C_7	req:I10 - C_{10}	req:I11 - C_1	req:I12 - C_3	req:I13 - C_4	req:I14 - C_8	req:I15 - C_9	req:I16 - C_{11}	req:I17 - C_{15}	req:I18 - C_{13}	req:I19 - C_{18}
CO				×	⊗	⊗		×		⊗	⊗	⊗	⊗	⊗	⊗	⊗		
PO				×	⊗	⊗		⊗	×	⊗	⊗	⊗	⊗	⊗	⊗	⊗		
AO1	×					×				⊗	⊗	⊗	⊗	⊗				
AO2	×					⊗	×			⊗	⊗	⊗	⊗	⊗				
SO		×			⊗	×				⊗	⊗	⊗	⊗		⊗	⊗	⊗	
SAO		×			⊗	⊗	×			⊗	⊗	⊗	⊗		⊗	⊗	⊗	
CDB																		
SDB																		
GDB																		

	pro:I1 - C_{12}	pro:I2 - C_{17}	pro:I3 - C_{14}	pro:I4 - C_{16}	pro:I5 - C_6	pro:I6 - C_2	pro:I7,I9 - C_5	pro:I8 - C_7	pro:I10 - C_{10}	pro:I11 - C_1	pro:I12 - C_3	pro:I13 - C_4	pro:I14 - C_8	pro:I15 - C_9	pro:I16 - C_{11}	pro:I17 - C_{15}	pro:I18 - C_{13}	pro:I19 - C_{18}
CO																		
PO																		
AO1																		
AO2																		
SO																		
SAO																		
CDB	×						×											⊗
SDB		×					×											⊗
GDB		⊗	⊗	⊗	×	×	⊗	⊗	⊗						⊗	⊗	⊗	⊗

5.2 *Usage of the lattice of component types*

While interfaces represent parts of collaborations, component types introduce consistent units dedicated to the provision of a consistent set of services. As in the previous lattices, but at a higher level in the structure of software artifacts, the lattice of component types offers both a specialization relation between component types and new abstract component types. This lattice has several applications in component assembly and software application re-engineering.

5.2.1 Emergence of new component types

The concepts in the lattice of component types can be interpreted as component types that we define as "canonical component types" to remain coherent with the previous definitions. Some of these canonical component types correspond to the original components: they are associated with concepts the reduced extent of which contains an original component. When the reduced extent of a concept is empty, we explore the intent of the concept to build the corresponding canonical component type. Thus, we consider symmetrically the required and provided interfaces from the intent. In the case of required interfaces, we consider those that have the smallest (more specific) type as shown in the interface lattice. In the case of provided interfaces, we consider those that have the largest (more general) type. These rules are a transcription of the substitution rules for functionality signatures, extended to interfaces.

Definition 5.3 Canonical component type corresponding to a component type concept**:** Let C be a concept in the component type lattice $\mathcal{L}_C$. The canonical component type $T_c(C)$ is defined as follows:

- If $ReducedE(C) \supseteq \{T\}$, then $T_c(C) = T$. We can choose any component type from the reduced extent because they are all equivalent.
- If $ReducedE(C) = \varnothing$, then $T_c(C) = \{pro : I, I \in max_{\leq_{\mathcal{L}_I}}\{J|pro : J \in Intent(C)\}\} \cup \{req : I, I \in min_{\leq_{\mathcal{L}_I}}\{J|req : J \in Intent(C)\}\}$.

In the case where an original component type appears in the reduced extent, the proposed construction finds an identical canonical component type. For example, concept C_{15} of lattice $\mathcal{L}_C$ has {`pro:I5, pro:I6`} as its canonical component type

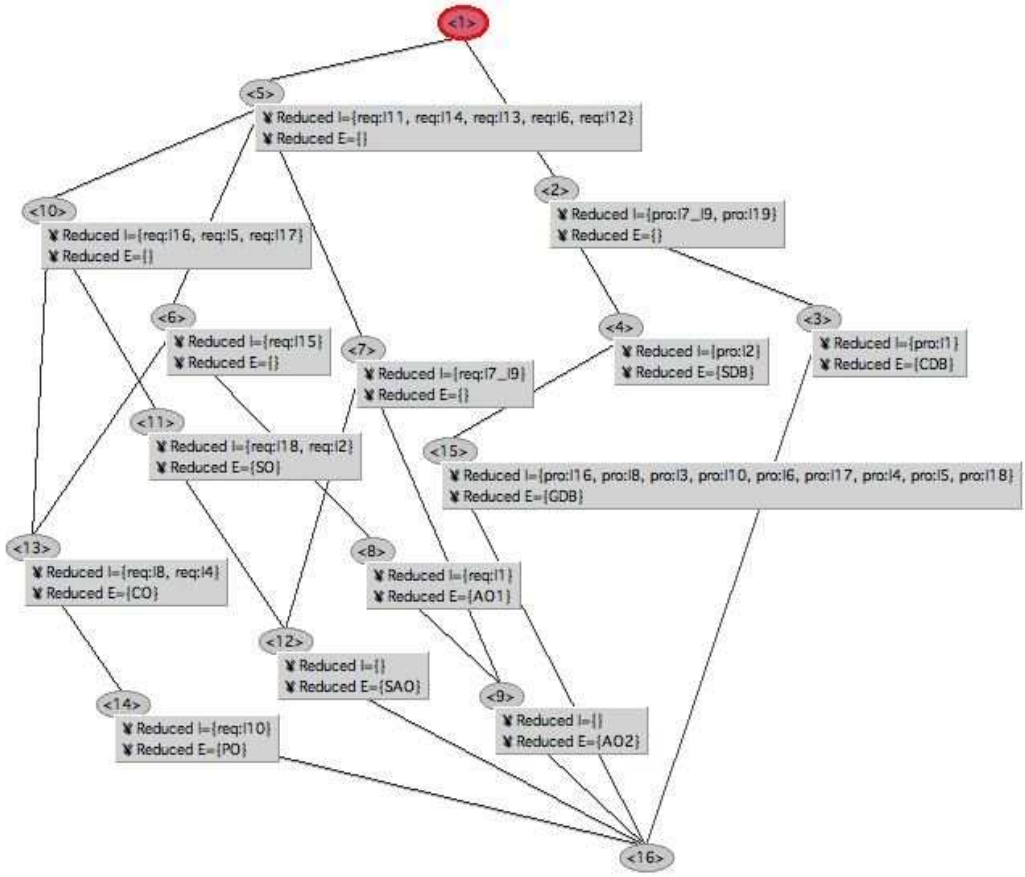

Figure 9. Lattice $\mathcal{L}_C$ of component types using the interface lattice.

because `I5` and `I6` are the maximum of $Intent(C_{15})$ (we do not make a distinction between required and provided interfaces because there are only provided interfaces in this intent). The reader will also notice that {`pro:I5, pro:I6`} is exactly the component type `GDB` that is found in the reduced extent of C_{15}.

5.2.2 Substitution and connection

The specialization relation we have built between concepts is tailored for substitution. Component substitution can be necessary in the event an entirely connected component fails. For example, let us suppose that an assembly is formed by component `CO` of type {`req:I8,req:I4`} entirely connected to component `GDB` of type {`pro:I5,pro:I6`}. Firstly, we can convince ourselves about the syntactical validity of the assembly that is ensured by two properties: required `I8` specializes provided `I6` and required `I4` specializes provided `I5` (as we generalize to interfaces the property described on Figure 5). Let us now imagine that component `CO` fails. Specialization in the lattice enables us to efficiently find a potential replacement. Component `PO` of type {`req:I10, req:I4`} will be a good candidate. The assembly remains valid because required `I10` specializes provided `I6`. The user will have access to a partial service because it is now only possible, among child books (`ChildBook` type), to ask for educational books (`EducationalBook` type), but the service may also perform better because it specializes about educational books.

Let us now analyze the connection problem. We note that two complementary components are not necessarily related to each other in the lattice: for example, there is no link between the components `AO2` of type {`req:I1,req:I7`} (concept C_9) and `CDB` of complementary type {`pro:I1,pro:I7`} (concept C_3). Indeed `req:I`

and `pro:I` are considered independent attributes. Given a component (*e.g.,* `AO2` of type {`req:I1,req:I7`}), it is nonetheless possible to find components that it can be connected to. A solution firstly consists in classifying the type of its complementary component (*e.g.,* {`pro:I1,pro:I7`}) applying the inferences. In our example, we obtain {`pro:I1,pro:I7,pro:I19`}. In this case, the classification enables us to reach concept C_3. C_3 and all smaller (more specific) concepts define, by the mean of their corresponding canonical component type, the types of components that can entirely connect to `AO2`.

5.2.3 Reingeneering and building generic architectures

We have previously described how the lattice discovers new component types. For example, concept C_5 of canonical type {`req:I14,req:I6`} has an empty extent. It indicates that the concept does not precisely correspond to an original component. However, it is an abstraction of all component types corresponding to lower (more specific) concepts. This canonical type, {`req:I14,req:I6`}, abstracts components relative to product orders in the example. It can be replaced by any of the more specific components. If a component of this canonical type participates in a component architecture, this architecture will have the capability of being instantiated using an important variety of concrete components. The discovery of such new abstract components into the classification can be interpreted as reengineering the set of existing components, and can help the developer design more generic architectures.

5.2.4 Architecture abstraction

The component lattice shows both specialization relationships among component types and newly discovered abstract component types. This can serve as the basis of whole architecture classification. This new objective is a little less direct to reach than the other generalization steps we have described in the paper because, in an architecture, components are not only described by binary attributes but also by their interconnections. Several ideas can be explored to take into account these connections such as Relational Concept Analysis (Huchard *et al.* 2007) or relations in Logical Information Systems (Ferré *et al.* 2005).

6. Related work

Few of the related approaches use a syntactical type hierarchy to structure component indexes and help component search. Zaremski and Wing (1995) suggest such a mechanism but in the more general context of functionality signature matching. The functionality hierarchy lies on the partial order relationship defined by the signature matching operator used, whether it be exact or relaxed. Module matching (component matching) is deduced from functionality matching: a component is comparable with another if each of its functionalities match a functionality of the other.

Existing yellow page-based service directories, also called service traders (Iribarne *et al.* 2004), such as CORBA Trading Object Service (OMG 2000), conform to the principles of the ODP standard (Information Technology Open Distributed Processing 1998). A component exports an advertisement into the component directory in order to be registered as the provider of some service. The service advertisement conforms to an existing service type that lists the properties and syntactical interfaces the components must have to provide the service. Service types can be ordered in a specialization hierarchy which is static and manually built. As opposed to our approach, these models use statically defined service

hierarchies (Marvie *et al.* 2001). This kind of indexing and the corresponding directories are not adapted to dynamical, evolving and open environments.

Works based on FCA propose to semi-automatically index components (Lindig 1995) in order to be able to help the developer identify adequate components from all the components stored in a component repository. Component search lies on groups of names and keywords and on incremental queries that help focus the search, diminishing the number of potential results as the search gets more precise. Fischer (1998), Sigonneau and Ridoux (2004) both aim at building such browsable functionality directories. Concepts are used to handle the iterative selection of attributes that define the user request as a traversal of the concept hierarchy. Thus, in these approaches, concept hierarchies do not directly reflect specialization relations between the syntactical types of functionality signatures. Fischer (1998) uses attributes which represent fragments of the formal specifications of functionalities (elementary pre- and post- conditions). Sigonneau and Ridoux (2004) use syntactical types of input and output parameters, along with covariant and contravariant specialization rules. In the context of web service search, machine learning techniques are used for service classification and annotation (Bruno *et al.* 2005, Corella and Castells 2006). Starting from textual documentation, services are automatically clustered using support vector machines or ontologies. FCA is then used in a second step to drive the matching between textual information and searched services.

As compared to these proposals, the originality of our work is to study directories of components described by sets of required and provided interfaces. Different specialization relations are defined to take into account not only parameter but also functionality directions. Moreover, we propose an iterative process to build lattices of component types which are composed of interfaces of both directions, which are in turn composed of functionalities. This iterative nature strongly differs from other works that use FCA which only build lattices of functionality types.

7. Conclusion and future work directions

In this article, we proposed to build component directories using FCA. The directory relies on the last built lattice that organizes components in order to speed up their retrieval, for either assembly or substitution. This component lattice is built upon some related lattices: an interface lattice which itself uses a classification provided by a functionality signature lattice. Beyond its usefulness for component assembly or component substitution, this classification also discovers new abstractions (new functionality signatures, new interface types and new component types), providing developers with valuable information about highly reusable elements. The developer can use this information as a guide along the development process or as re-engineering information.

The work presented in this article raises new research issues. Firstly, we want to study how our system can be implemented and integrated into an IDE to assist the management of component-oriented applications. This task comprises four steps:

- Extracting information about the component interfaces. We want to use the introspection capabilities of components to extract and dynamically maintain information on interfaces as the components enter or leave the system.
- Encoding the information in formal contexts, taking into account the identified inference rules and the type hierarchy of the system.

- Building lattices. Kuznetsov and Obiedkov (2002) present several incremental algorithms that enable new concepts to be added to an existing lattice. Several of these algorithms are implemented in GaLicia (Valtchev *et al.* 2003). These algorithms could be used to calculate the different lattices and also maintain them dynamically as components enter or leave the system.
- Using lattices. The obtained lattices can be used not only as a component index to ease search, but also as a way of visualizing the content of component libraries using the graphical interface of GaLicia or a similar FCA tool like TOSCANA (Vogt and Wille 1994) or CONEXP (Yevtushenko 2000).

This will enable us to systematically experiment with our approach on large component repositories, considering various component or interface granularity and function signature complexity.

We also plan to study complementary features of components, interfaces and signatures, such as ports, protocols or exceptions. For instance, ports (Desnos *et al.* 2006, 2007) would enable specifications of the dynamic behavior of components to be considered, providing more accurate component indexing and retrieval.

Another extension is inspired by Web Services directories (Klusch 2008). Contrary to component directories, they mainly use semantic information (names, descriptions) in their search mechanism. We can experiment with these techniques to refine classification considering the name of the parameters in the functionality signatures. Conversely, it is interesting to analyze how our approach could be used to improve the calculation of syntactical compatibility in Web Services.

Acknowledgments

Authors which to thank Nour Alhouda Aboud for her careful reading of a draft version of this paper. They also want to thank Peter W. Eklund, the guest editors and the anonymous reviewers of the paper for their helpful comments.

References

Birkhoff, G., 1940. *Lattice theory.* AMS Colloquium Publication, vol. XXV.

Bruno, M., Canfora, G., Penta, M.D. and Scognamiglio, R., 2005. An Approach to support Web Service Classification and Annotation. *In*: W.K. Cheung and J. Hsu, eds. *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)* Washington, DC, USA: IEEE Computer Society, 138–143.

Cardelli, L., 1984. A Semantics of Multiple Inheritance. *In*: G. Kahn, D.B. MacQueen and G.D. Plotkin, eds. *Semantics of Data Types*, LNCS 173 Springer, 51–67.

Corella, M.Á. and Castells, P., 2006. Semi-automatic Semantic-Based Web Service Classification. *In*: J. Eder and S. Dustdar, eds. *Business Process Management Workshops*, LNCS 4103 Springer, 459–470.

Davey, B.A. and Priestley, H.A., 1991. *Introduction to lattices and orders.* second Cambridge University Press.

Desnos, N., Huchard, M., Tremblay, G., Urtado, C. and Vauttier, S., 2008. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution: Research and Practice. Special Issue on Search-Based Software Engineering*, 20 (5), 321–344.

Desnos, N., Huchard, M., Urtado, C., Vauttier, S. and Tremblay, G., 2007.

Automated and unanticipated flexible component substitution. LNCS 4608 Springer, 33–48.

Desnos, N., Vauttier, S., Urtado, C. and Huchard, M., 2006. Automating the Building of Software Component Architectures. LNCS 4333 Springer, 228–235.

Ferré, S., Ridoux, O. and Sigonneau, B., 2005. Arbitrary Relations in Formal Concept Analysis and Logical Information Systems. *In*: F. Dau, M.L. Mugnier and G. Stumme, eds. *ICCS 2005*, LNCS 3596 Springer, 166–180.

Fischer, B., 1998. Specification-based Browsing of Software Component Libraries. *In*: D.F. Redmiles and B. Nuseibeh, eds. *Proceedings of the 13th IEEE international conference on Automated Software Engineering (ASE'98)*, October. Honolulu, Hawaii, USA: IEEE Computer Society, 74–83.

GaLicia, Galois lattice interactive constructor. : Université de Montréal. http://www.iro.umontreal.ca/∼galicia - accessed on Sept. 22, 2008 [2002].

Huchard, M., Hacene, M.R., Roume, C. and Valtchev, P., 2007. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49 (1-4), 39–76.

Information Technology Open Distributed Processing, ODP Trading Function Specification ISO/IEC 13235-1:1998(E). : International Organization for Standardization and International Telecommunication Union. http://webstore.iec.ch/preview/info_isoiec13235-1%7Bed1.0%7Den.pdf - accessed on Sept. 22, 2008 [1998].

Iribarne, L., Troya, J.M. and Vallecillo, A., 2004. A Trading Service for COTS Components. *The Computer Journal*, 47 (3), 342–357.

Klusch, M., 2008. Semantic service coordination. *In*: M. Schumacher, H. Helin and H. Schuldt, eds. *CASCOM - Intelligent Service Coordination in the Semantic Web*. Birkhaeuser Verlag, Springer, chap. 4.

Kuznetsov, S.O. and Obiedkov, S.A., 2002. Comparing performance of algorithms for generating concept lattices.. *Journal of Experimental & Theoretical Artificial Intelligence*, 14 (2-3), 189–216.

Lindig, C., 1995. Concept-Based Component Retrieval. *In*: J. Köhler *et al.*, eds. *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 21–25.

Marvie, R., Merle, P., Geib, J.M. and Leblanc, S., 2001. Type-Safe Trading Proxies Using TORBA. Fifth International Symposium on Autonomous Decentralized Systems, ISADS, IEEE Computer Society, 303–310.

Meyer, B., 1991. Design by contract. *In*: D. Mandrioli and B. Meyer, eds. *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1–50.

OMG, Trading Object Service Specification (TOSS) v1.0. : Object Managment Group. http://www.omg.org/cgi-bin/doc?formal/2000-06-27 - accessed on Sept. 22, 2008 [2000].

Sigonneau, B. and Ridoux, O., 2004. Indexation multiple et automatisée de composants logiciels orientés objet. *In*: J. Julliand, ed. *AFADL — Approches Formelles dans l'Assistance au Développement de Logiciels*, Juin. Besançon, France: RTSI, Lavoisier.

Szypersky, C., Gruntz, D. and Murer, S., 2002. *Component Software - Beyond Object-Oriented Programming (Second Edition)*. Second Addison-Wesley / ACM Press.

Valtchev, P., Grosser, D., Roume, C. and Hacene, M.R., 2003. GaLicia: An Open Platform for Lattices. *In*: A. de Moor, W. Lex and B. Ganter, eds. *Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03)*, Dresde (DE), July. Http://www.iro.umontreal.ca/∼galicia Shaker Verlag, 241–254.

Vogt, F. and Wille, R., 1994. TOSCANA - a Graphical Tool for Analyzing and

Exploring Data. *In*: R. Tamassia and I.G. Tollis, eds. *Graph Drawing*, LNCS 894 Springer, 226–233.

Wille, R., 1982. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, 83, 445–470.

Yevtushenko, S.A., ConExp, http://conexp.sourceforge.net/index.html. : SourceForge. [2000].

Zaremski, A.M. and Wing, J.M., 1995. Specification matching of software components. *In*: G.E. Kaiser, ed. *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, Washington, D.C., United States, October. New York, NY, USA: ACM Press, 6–17.

Zaremski, A.M. and Wing, J.M., 1997. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6 (4), 333–369.