Precalculating Component Interface Compatibility Using FCA

Gabriela Arévalo[1], Nicolas Desnos[2], Marianne Huchard[3], Christelle Urtado[2], and Sylvain Vauttier[2]

[1] LIFIA - Facultad de Informática (UNLP) - La Plata - Argentina
garevalo@sol.info.unlp.edu.ar
[2] LGI2P / Ecole des Mines d'Alès - Nîmes - France
{Nicolas.Desnos, Christelle.Urtado, Sylvain.Vauttier}@ema.fr
[3] LIRMM - CNRS and Univ. Montpellier 2 - Montpellier - France
Marianne.Huchard@lirmm.fr

Abstract. In component-based software engineering, software systems are built by assembling prefabricated reusable components. The compatibility between the assembled components is crucial. It is determined by the comparison of their exposed interfaces: required interfaces (describing the services the component needs) and provided interfaces (describing the services the other component offers) must match. Given a component, finding a compatible component in a component repository is not trivial. The idea of this paper is that organizing component directories with a yellow-page-like structure makes the search for suitable components more efficient. We propose a solution based on Formal Concept Analysis to precalculate a concept lattice to organize our components. It proves to be an efficient solution to both represent the component compatibility information and provide a browsable component organization to support the component search mechanism.

1 Introduction

The component-based approach is a recent and successful paradigm for software engineering, inspired by electronic engineering. In this approach, software systems are built by assembling prefabricated reusable components. The main objective is development cost reduction while maintaining high quality. Software components are externally described, as their electronic counterpart, by functionalities they support, and plugs which specify possible connections. In terms of the component-based software engineering (CBSE) domain, a component has required interfaces (needed services) and provided interfaces (offered services). Building a component assembly consists in connecting components in order to achieve a high-level functionality. Computation is then dispatched over the assembled components. Nevertheless, given a component repository, finding and connecting suitable components is not trivial because there is basically a need to determine service compatibility. Component repositories generally rely on component directories organized as white pages, that support component search by

name. Such an organization is not optimal because it leaves the burden of finding suitable components to the assembly phase.

Previous work on component assemblies [1, 2] made us realize how critical the component-search mechanism was for such software engineering processes as component assembly building or evolution. The idea of this paper is that organizing component directories with a yellow-page-like structure makes the search for suitable components more efficient. Our approach is based on Formal Concept Analysis (FCA), where precalculating a lattice of service compatibilities is a means to organize the component directory so as to both explicitly represent relationships among components and efficiently search for suitable components when needed. Having a user-readable component structure that shows component compatibility is a plus and separating the concerns of compatibility calculus and assembly is more rational. Such an organized component directory can then be used to find a component that can assemble to (search for a compatible component) or replace (search for a substitutable component) a selected one. It also enables the discovery of new abstract components having a higher degree of reusability which enriches the component directory.

The remainder of this paper is organized as follows. Using a small sale system example, Section 2 shows how reasoning about component compatibility is mainly based on functionality signatures. Section 3 describes an extension of object-oriented type theory for functionality signature comparison. Then, after recalling chosen basics of FCA, Section 4 shows the building of a lattice of functionality signatures using the point of view of replacing a required functionality by another. Section 5 presents several uses of the proposed lattice, Section 6 compares our approach to related work and Section 7 mentions future work.

2 Component Compatibility: a Sale System Example

Service and functionality compatibility, as an *a priori* requirement for component compatibility, is discussed based on elements of a sale management application. Figure 1 shows several connection situations. When a component (*e.g.,* `ChildOrder`) is introduced in an assembly to satisfy a service (*e.g.,* enabling children to order products), the required interfaces involved in the same collaboration (here, `CustomerCreation`) have to be connected in the assembly. This may demand finding a compatible component either among those of the assembly, or in a component repository.

Component compatibility is generally defined as the syntactic comparison between the components' interfaces, which more precisely consists in comparing pairs of functionality signatures from these interfaces. To easily understand the discussion, the reader can draw a parallel with the ordinary function-call mechanism: required functionalities can be seen as function calls while provided functionalities are function definitions. Functionalities are described by their signatures, *i.e.,* their name, parameter type list and return type. To be run, a function needs its call to contain the expected information, as declared in its signature. The position, where the call appears in the code, expects in return a resulting

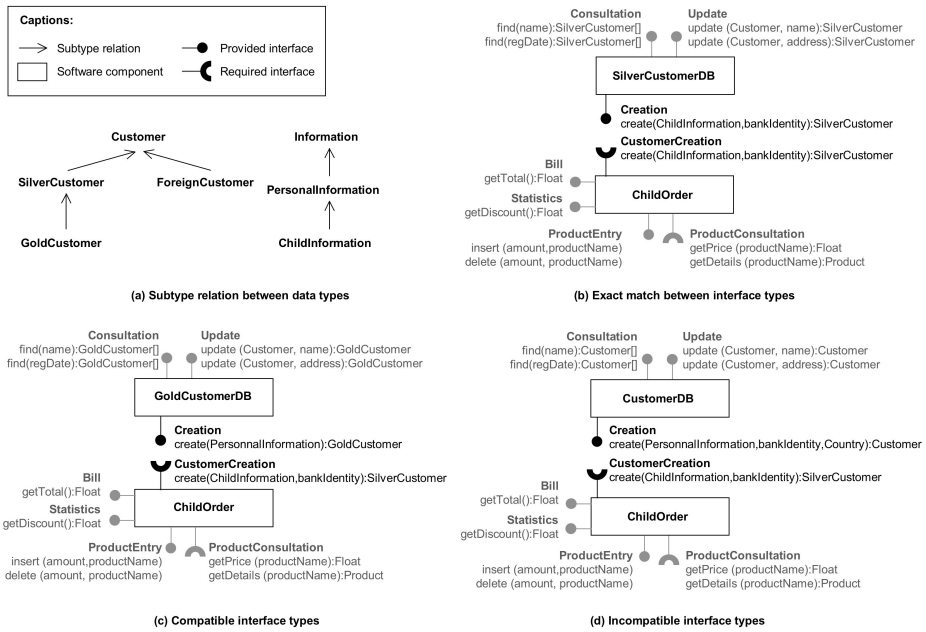

Fig. 1. Type ordering information and three possible assembly situations

data corresponding to the declared return type of the function. Figure 1(b) illustrates the simplest assembly case: the `SilverCustomerDB` component provides an interface which describes a `create` functionality that has exactly the same signature as the `create` functionality of the required `CustomerCreation` interface of the `ChildOrder` component – that is the same parameter type list and the same return type. However, exact signature match is not always possible: the component repository may only contain components with close capacities. In Figure 1(c), the `GoldCustomerDB` component offers a close provided creation functionality which turns out to be compatible with the one required by the `ChildOrder` component. There are three differences between the two signatures:

- The provided `create` functionality needs information about a person. As shown on Figure 1(a), `ChildInformation` is a subtype of `PersonnalInformation`. This means that a child is described by more information than a person. As a consequence, the required functionality is able to call the provided functionality as the call contains more information than needed by the provided functionality to run.
- The provided `create` functionality only needs information about a person: it has no other parameter. The fact that the required `create` gives extra information (an extra `bankIdentity` parameter is passed to the functionality call) does not hamper the assembly.
- The provided `create` functionality returns a parameter of type `GoldCustomer`. As shown in Figure 1(a), `GoldCustomer` is a subtype of `SilverCustomer`.

As in the other cases, the required `create` can still call the provided functionality as it will receive in return more information than needed.

In these three cases, the extra information provided during the call or the return phases can be ignored: the components can be assembled. Figure 1(d) presents a case of functionality incompatibility. The `CustomerDB` component provides a `create` functionality that is not compatible with the `create` functionality of the `ChildOrder` component. Two signature differences make the assembly impossible:

- The provided `create` needs information about the country of the customer: the required `create` is unable to provide such information.
- The provided `create` returns a parameter of type `Customer`. As shown in Figure 1(a), `Customer` is a supertype of `SilverCustomer`. The required `create` therefore receives in return an object which is too general and contains less information than expected.

This small example illustrates the need for signature comparison based on a model of component compatibility, where parameter and return types have a great place. In realistic examples, an interface generally contains more than one functionality signature. This is why reasoning on component compatibility requires to have a theory of functionality signature compatibility, which can be extended to nesting (interface, component) levels. As shown in the example, signature compatibility is the result of the combination of subtyping orders on parameter and return types as well as presence / absence of parameters. In the case where the component assembler is human, mentally calculating this combination is rather difficult, increasing the errors when choosing a component among the available ones. Besides, a mental image, by the means of a classification of signatures, could be quite useful. At the same time, this could give us keys for helper tools that guide the human designer in its component choice, or even, in simple cases, that solve the choice problem automatically.

Thanks to its qualities for building classifications of entities, FCA is very useful to cope with this problem. The produced signature classifications can be used either to check compatibility if a component is available, to find a component (using the classification as an index for component access) or to build contingent interface or component classifications by propagating specialization information. Thanks to FCA, signature classification is not reduced to the organization of existing signatures: new signatures emerge that are generalizations (abstractions) of the existing ones, giving the opportunity to the component developer to imagine new components that would be more reusable than the existing ones.

3 Basics of Static Types: Ordering Required and Provided Functionalities

This section focuses on presenting the domain knowledge about signatures that will be encoded into concept lattices in order to capture functionality substi-

tutability information. Part of assembly soundness is based on static types, inspired by statically typed object-oriented languages [3]. In this context, static types are used to guide an efficient analysis that considerably limits dynamic type errors potentially provoked by the intensive usage of polymorphism. In object-oriented languages, polymorphic expressions can be bound to values of several types (classes in this context). The correctness of this binding relies on the possibility for an expression of a type to be seamlessly replaced (substituted) by any value of another type. This substitutability is possible under some sufficient conditions on types relative to their operations: the run-time type error that static analysis wants to avoid is the reception by an object of an operation request that it cannot deal with. In the object paradigm, when an operation is redefined in a subclass (subtype), type-safe definitions of operations respect two constraints:

- *Parameter type contravariance.* A parameter type has to vary in the opposite direction as subclassing: in the subclass, the parameter has a more general type (super-type) than in the superclass.
- *Return type covariance.* The return type has to vary in the same direction as subclassing: in the subclass, the return type has a more specialized type (subtype) than in the superclass.

Let us study how this theory extends to the component domain. In order to define component substitutability, we only consider the smallest abstractions (functionalities) because the whole problem reduces to determining if a component that provides (*resp.* requires) a functionality can replace a connected component that provides (*resp.* requires) another functionality. Functionalities are described by their *name* (replacement is admitted only between same name functionalities), *parameter type list* (IN parameter types), and *return types* (OUT parameter types). Rules for substitutability are as follows for required functionalities (rules for provided functionalities are obtained by a symmetric reasoning):

- *IN parameter type specialization.* If a required functionality is able to send a parameter of some type, it can replace a required functionality which sends a parameter of a more general type (because the called functionality can ignore the information specific to the specialized type). In the example of Figure 2(a), required functionality `create(PersonalInformation)` can replace required functionality `create(Information)` because in the context where required `create(Information)` can be connected (namely, to provided `create(Information)`), required `create(PersonalInformation)` can also be connected as it can connect to provided `create(Information)` and to provided `create(PersonalInformation)` which is a superset.
  This is an application of the substitutability rule in object-oriented languages and obeys the contravariance principle (because in the provided point of view, IN parameter types need generalization).
- *IN parameter addition.* If a required functionality is able to send a parameter of a particular type, it can replace a required functionality which does not send this kind of parameter (because the called functionality can ignore this

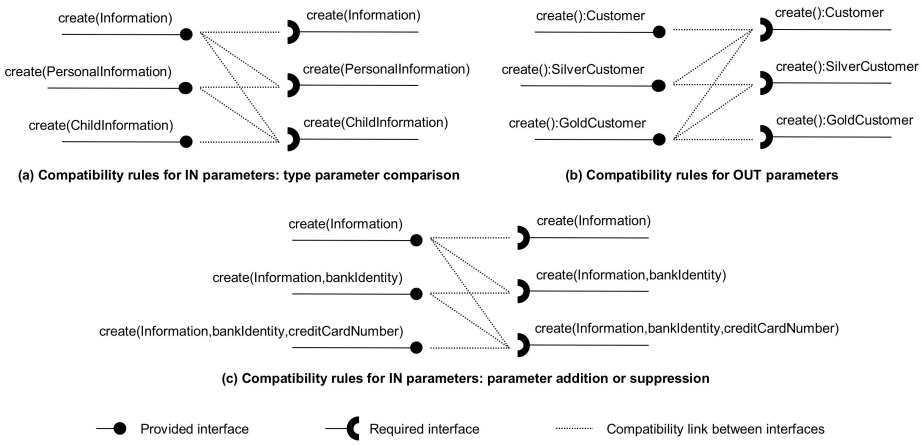

Fig. 2. Interface compatibility rules when parameter types and number vary

parameter). For example (Figure 2(c)), required `create(Information,bankIdentity)` can replace required `create(Information)` because in the context where required `create(Information)` can be connected, required `create(Information,bankIdentity)` can be connected too.

– *OUT parameter type specialization.* If a required functionality needs to receive some return type, it can replace a required functionality which needs to receive a more specialized type (as the extra information in the specific type can always be ignored). For example (Figure 2(b)), required `create():Customer` can replace required `create():SilverCustomer` because in the context where required `create():SilverCustomer` can be connected, required `create():Customer` can be connected too. This is another application of the substitutability rule in object-oriented languages and obeys the covariance principle (also considering the provided point of view).

4 Lattice of Functionality Compatibilities

This section firstly recalls the basics of FCA and then presents the building principles of the functionality signature lattice.

4.1 A Survival Kit for Formal Concept Analysis

The classification we build is based on the partially ordered structure known as *Galois lattice* [4] or *concept lattice* [5] which is induced by a context K, composed of a binary relation R over a pair of sets O (*objects*) and A (*attributes*) (Figure 3).

A concept C is a pair of corresponding sets X and Y such that:

$X = \{ x \in O | \forall y \in Y, (x, y) \in R\}$ is called *extent* (covered objects)
$Y = \{ y \in A | \forall x \in X, (x, y) \in R\}$ is called *intent* (shared features)

For example, $(12, bc)$[4] is a formal concept, but $(2, bc)$ is not. Establishing that $(12, bc)$ is a concept highlights the fact that objects 1 and 2 exactly share attributes b and c (and vice-versa). Furthermore, the set of all concepts $\mathcal{C}$ constitutes a lattice $\mathcal{L}$ when provided with the following specialization order based on intent / extent inclusion: $(X_1, Y_1) \leq_{\mathcal{L}} (X_2, Y_2) \Leftrightarrow X_1 \subseteq X_2$ (or equivalently $Y_2 \subseteq Y_1$). Figure 4 shows the Hasse diagram of $\leq_{\mathcal{L}}$.

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

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

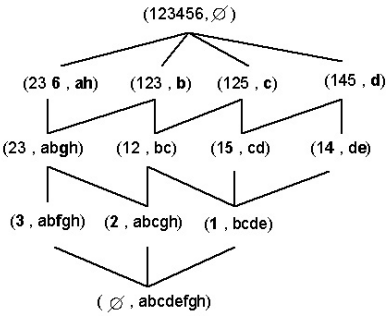

Fig. 4. Concept lattice $\mathcal{L}$.

4.2 Building the 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 (PersonalInformation,bankIdentity):SilverCustomer` and comparing it to `create (Information):GoldCustomer`. At first, for each functionality, attributes are deduced from IN and OUT parameter types that explicitly appear in the signature. These attributes are marked using × in Figure 5. Then, we infer attributes (marked with ⊗ in Figure 5) 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 also implicitly sends a parameter of any more general type (because the called provided functionality can ignore the part of the information that is specific to the specialized type).
- 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 (the received extra information can be ignored).

In component-based systems, all the information about the functionalities described in (provided or required) component interfaces is available at runtime.

[4] $(12, bc)$ is the compact notation for concept $\{1, 2\} \times \{b, c\}$.

Indeed, thanks to reflexivity (based on either metadata or introspection), a component can be asked each detail of one of its functionality signature (name of the functionality, types of IN and OUT parameters, etc.) at runtime. This capability could be used as the basis of an automatic process that would build the binary context and identify the inferred attributes.

Figure 6 depicts the lattice corresponding to the binary relation shown in Figure 5, built with the Galicia tool [6]. Concepts are presented using reduced intents and extents for readability sake: an object (signature) which belongs to the reduced extent of a concept is inherited by all concepts that are above (down-to-up); a property (IN or OUT parameter type) which belongs to the reduced intent of a concept is inherited by all concepts that are below (up-to-down).

The following section gives insights of the different operations this kind of lattice can support for the management of component directories.

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

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

Fig. 5. Encoding of a set of required functionality signatures

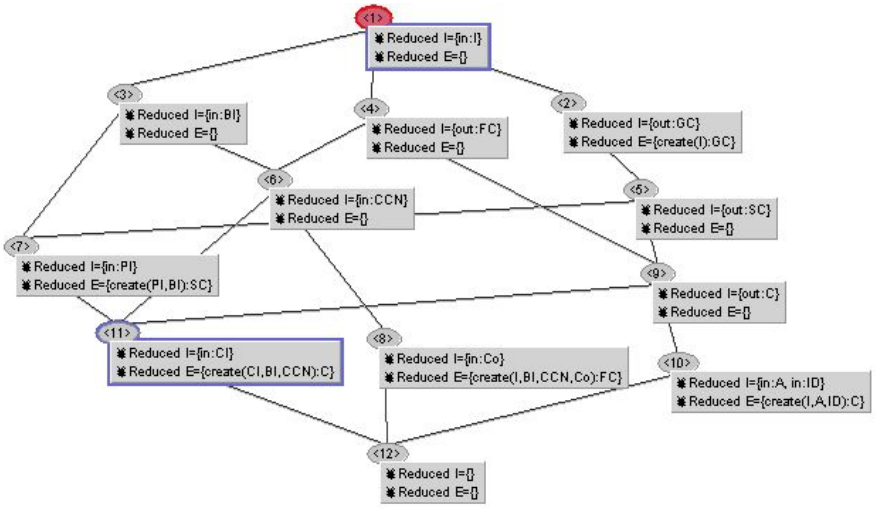

Fig. 6. Lattice $\mathcal{L}_s$ of service signatures

5 Using the Lattice of Functionality Compatibility

The lattice of signatures supports three main usages. Firstly, it determines which functionality can replace which other, in both cases of required or provided functionalities (Section 5.1). Secondly, the validity of the connection of a required functionality to a provided functionality can be stated (Section 5.2). Finally, in a prospective section, it is used as the starting point of a construction chain that goes from functionality signature to component type classification (Section 5.3).

5.1 Required/ Provided Functionality Substitution

Consider the lattice of Figure 6 with the viewpoint of required functionalities. In this lattice `create(I):GC` is represented by Concept 2 while `create(PI,BI):SC` is represented by Concept 7. Concept 2 is more general than Concept 7 which can be interpreted as Concept 7 can replace Concept 2. In a component architecture, a connection to a required functionality corresponding to Concept 2 can be replaced by a connection to a required functionality corresponding to Concept 7. In the general case, when there is a path between two concepts, the more specific one (which has more properties) can replace the more general one (which has a subset of properties) when the more general concept is connected (see Figure 7(a)).

The same lattice can also be used for provided functionality substitution if read in the reverse order (see Figure 7(b)). In the previous example, symmetric assertion can be made: in situations where provided functionality corresponding to Concept 7 are connected, provided functionality corresponding to Concept 2 can also be connected. Next rule formalizes how functionality substitution can generally be deduced from the lattice.

Substitution rule. Let C_{father}, C_{son} be two concepts of the functionality signature lattice such that $C_{son} \leq_{\mathcal{L}_s} C_{father}$. Functionalities of C_{son} can replace functionalities of C_{father} when the functionalities are required. Opposite replacement applies when the functionalities are provided.

5.2 Inferring Valid Connections

Both points of view (provided and required) can be combined to address the issue of component connection. Consider the signature `create(PI,BI):SC`. Required `create(PI,BI):SC` evidently can connect to provided `create(PI,BI):SC`. Given the substitution rule, provided functionalities which are upper in the lattice, for example provided `create(I):GC`, can be connected to required `create(PI,BI):SC` (see Figure 7(c)). Using the same rule in the symmetric case, required functionalities which are below in the lattice, for example required `create(CI,BI,CCN):C`, can be connected to provided `create(PI,BI):SC`. By transitivity, required `create(CI,BI,CCN):C` can be connected to provided `create(I):GC`. This is expressed in the following connection rule that formalizes how valid functionality connection can be deduced from the lattice.

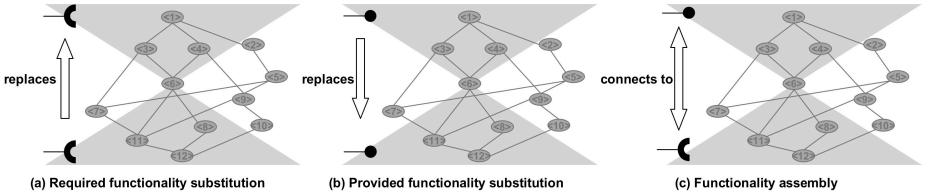

Fig. 7. Interpretation of the lattice of service signatures

Connection rule. Let C, C_{father}, C_{son} be three concepts of the functionality signature lattice such that $C_{son} \leq_{\mathcal{L}_s} C \leq_{\mathcal{L}_s} C_{father}$, required functionalities of C_{son} can be connected to provided functionalities of C_{father}.

5.3 Towards Component Classification

Although several component models exist for CBSE, most of them include the notions of interface and functionality. Components are reusable pieces of software that can be chosen off-the-shelf and have a high-level objective (database component, counter component, scheduler component, etc.). Interfaces group functionalities that form meaningful collaborations together and have a direction (provided or required).

Functionality classification is the starting point of a classification chain that ends in component classification, using a non-iterative version of Relational Concept Analysis [7]. We detail only the level of interfaces to give an idea of the process. An interface is described by a set of functionalities. Therefore, a formal context for interfaces naturally associates functionalities with interfaces. However, to benefit from knowledge acquired in the lattice of service signatures, a richer description of interfaces is obtained using the concepts of this lattice as formal attributes. For example, let us consider two (required) hypothetic interfaces: `SilverCustomerDBcreation` which includes `create(PI,BI):SC` and `ChildDBcreation` which includes `create(I,A,ID):C`. As `create(PI,BI):SC` is in Concepts 7, 5, 2 and 1, these concepts are used as formal attributes for interface `SilverCustomerDBcreation`. Similarly, `create(I,A,ID):C` appears in Concepts 10, 9, 5, 2, 4 and 1, giving part of the formal attributes of interface `ChildDBcreation`. This kind of encoding corresponds to existential scaling operator of the RCA approach [8]. Based on a discovered functionality of the first lattice (`create(I):SC` in Concept 5), the technique can infer a new interface, including at least this shared functionality . Here is a fundamental advantage of FCA/RCA techniques compared to the simple computing of signature comparison: new signatures emerge that provoke emergence of new interfaces which abstract the existing ones, etc. The process has to further be tuned by the direction of interfaces (provided versus required) which was not included in the above discussion for simplicity sake and reverses the specialization order on signatures.

The next generalization level concerns components. Discovering new component external specifications, in the final step of the process is of great interest for

component designers because they discover abstract, more reusable components that can guide them in their development work.

6 Related Work

A service trader (yellow page mechanism) is a kind of directory that allows to index and locate components through the definition of the services they provide [9]. Existing proposals, such as CORBA Trading Object Service [10], mostly follow the ODP standard principles [11]. A component exports to the directory a service advertisement to be registered as a provider of a service. The service advertisement conforms to a service type that specifies the properties and the syntactic interfaces that a component must feature to provide this service. Service types are organized as a specialization hierarchy. Requests are sent to the directory to look for providers of some service, as defined by its type. Corresponding service advertisements can be filtered by conditions defined on property values.

Previous works do not use FCA as a base methodology. In this kind of approaches, a semi-automatic indexing methodology is proposed in [12] to help the developer identify suitable components in an existing repository. The retrieval is based on grouping names and keywords, and incremental queries that help to refine the search of a component. Within the context of identifying web services, the approaches are based on machine learning techniques, to support service classification and annotation [13, 14]. Starting from free text service documentation, services are automatically classified in classes/domains using Support Vector Machines or Ontologies. Successively, FCA is used to group text-based information related to the components to be identified. Contrary to our proposal, these approaches (some of which use FCA and some not) use an explicitly and statically constructed service type hierarchy [15]. Service type hierarchy and syntactic type hierarchy are only informally related. Moreover, only information about provided services and interfaces are considered. This content definition and organization limits the usages of this kind of component directories in dynamic, evolving and open environments.

7 Conclusions and Future Work

Applications of FCA, besides their contribution to a specific domain, participate in constructing a shared expertise about FCA-based methodologies. The contribution of this paper mainly focuses on technical aspects of encoding into formal contexts and on the definition of relevant usages of the built lattice in identifying compatible components. The main difficulty we encountered when encoding was to capture type specialization in opposite ways depending on parameter (IN / OUT) and functionality (provided / required) directions. The gain of FCA for functionality signature classification consists in proposing a classification where not only substitution and connection can be read, but also in which new, more abstract, signatures appear. These new functionality signatures

enable the construction of component directories via interface and component classifications.

Perspectives include integrating complementary features of components as proposed in some component models [16, 17, 2]. Further research will also consider fully integrating such a pre-calculation in a yellow page components mechanism. Incrementally building the lattice will then be an extra advantage to manage incoming and outgoing components in a very dynamic environment.

References

1. Desnos, N., Vauttier, S., Urtado, C., Huchard, M.: Automating the building of software component architecture. In Gruhn, V., et al., eds.: Software Architecture: 3^{rd} EWSA Workshop. Vol. 4344 of LNCS, France, Springer (2006) 228–235
2. Desnos, N., Huchard, M., Urtado, C., Vauttier, S., Tremblay, G.: Automated and unanticipated flexible component substitution. In Schmidt, H.W., et al., eds.: Proc. of 10^{th} ACM SIGSOFT CBSE. Vol. 4608 of LNCS, USA, Springer (2007) 33–48
3. Cardelli, L.: A semantics of multiple inheritance. Vol. 173 of LNCS., Berlin, Springer-Verlag (1984) 51–64
4. Barbut, M., Monjardet, B.: Ordre et Classification. Hachette (1970)
5. Wille, R.: Restructuring lattice theory: an approach based on hierarchies of concepts. Ordered Sets **83** (1982) 445–470
6. GaLicia: Galois lattice interactive constructor Université de Montréal. http://www.iro.umontreal.ca/∼galicia.
7. Dao, M., Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Improving generalization level in UML models iterative cross generalization in practice. In Wolff, K.E., et al., eds.: ICCS. Vol. 3127 of LNCS, USA, Springer (2004) 346–360
8. Huchard, M., Rouane Hacene, M., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Annals of Mathematics and Artificial Intelligence **49**(1-4) (2007) 39–76
9. Iribarne, L., Troya, J.M., Vallecillo, A.: A trading service for COTS components. The Computer Journal **47**(3) (2004) 342–357
10. OMG: Trading Object Service Specification v1.0. (2000)
11. International Organization for Standardization and International Telecommunication Union: ISO/IEC 13235, ITU-T X.9tr, Information Technology Open Distributed Processing ODP Trading Function. (1996)
12. Lindig, C.: Concept-based component retrieval. In Köhler, J., et al., eds.: Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs. (1995) 21–25
13. Bruno, M., Canfora, G., Penta, M.D., Scognamiglio, R.: An approach to support web service classification and annotation. In: Proc. of the EEE'05 Conference, USA, IEEE Computer Society (2005) 138–143
14. Corella, M.A., Castells, P.: Semi-automatic semantic-based web service classification. In: International Workshop on Advances in Semantics for Web Services, Springer Verlag (2006) 459–470
15. Marvie, R., Merle, P., Geib, J.M., Leblanc, S.: Type-safe trading proxies using TORBA. In: ISADS. (2001) 303–310
16. OMG: Unified modeling language: Superstructure, version 2.0 (2002) http://www.omg.org/uml.
17. Bures, T., Hnetynka, P., Plásil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: SERA, IEEE Computer Society (2006) 40–48