Automated architectural component classification using concept lattices

Nour Alhouda Aboud*, Gabriela Arévalo†, Jean-Rémy Falleri‡, Marianne Huchard‡, Chouki Tibermacine‡
Christelle Urtado§ and Sylvain Vauttier§
*LIUPPA, Univ. de Pau et des Pays de l'Adour, Bayonne, France
Email: n.aboud@etud.univ-pau.fr
†LIFIA, Facultad de Informática (UNLP), La Plata, Argentina
Email: Gabriela.Arevalo@lifia.info.unlp.edu.ar
‡LIRMM, CNRS and Montpellier II University, France
Email:{Jean-Remy.Falleri, Marianne.Huchard, Chouki.Tibermacine}@lirmm.fr
§LGI2P, Ecole des Mines d'Alès, Nîmes, France
Email: {Christelle.Urtado, Sylvain.Vauttier}@ema.fr

Abstract

While the use of components grows in software development, building effective component directories becomes a critical issue as architects need help to search components in repositories. During the life-cycle of component-based software, several tasks, such as construction from scratch or component substitution, would benefit from an efficient component classification and retrieval. In this paper, we analyze how we can build a classification of components using their technical description (i.e. functions and interfaces) in order to help automatic as well as manual composition and substitution. The approach is implemented in the CoCoLa prototype, which is dedicated to Fractal component directory management and validated through a case study.

1. Introduction

Component-based software engineering promotes reuse in the large: off-the-shelf software components are assembled to build complex applications. Such an assembly process is possible thanks to the availability of components' external description: required and provided interfaces syntactically describe the functionalities a component needs to find in its environment or provides to other components of its environment. This assembly process can be implemented at various stages of a software lifecycle. When designing software from scratch, the software architect needs to find software component types to assemble them before their instantiation and deployment. At runtime, autonomous software might self-assemble itself using available components. When evolving some existing software, a designer might also need to assemble some component types to an existing incomplete software design made from components. Finally, at runtime, dynamic autonomous software might need to re-assemble some of its parts to react to component failure or unavailability. During previous work on automatic component assembly and on dynamic component assembly evolution [1], we recognized that an efficiently indexed component directory was a central issue for component reuse.

This paper proposes an indexing mechanism for components that relies on type-theory and uses Formal Concept Analysis (FCA) [2] to build various specialization lattices that both offer human-readable views and computer-browsable indexes to search for suitable components to assemble or substitute to given ones. This indexing mechanism extends our previous proposal [3] with richer substitution semantics. Additionally, this paper describes the implementation of this indexing mechanism in the CoCoLa[1] tool, that could serve as the basis of an automatically built and search-oriented yellow-page component directory. The CoCoLa tool implements the automatic analysis and classification of Fractal [4] components. It is based on a three-step classification process that uses the syntactical description of functionalities and interfaces and the external view of components to iteratively classify these artifacts.

The structure of the paper is as follows. Section 2 motivates our work describing a didactic example we use throughout the paper. Section 3 provides an overview of the three-step classification process implemented in CoCoLa. Sections 4, 5 and 6 successively describe the three functionality, interface and component classification steps that use FCA to organize these artifacts into specialization hierarchies. Section 7 sketches the design of the CoCoLa component concept lattice building tool and describes experiments on a case study. Section 8 compares our approach to related work and Section 9 concludes.

2. Motivating Example

To explain our approach, we use a component repository that contains four concrete (implemented) components that implement various route calculation algorithms. Figure 1

1. CoCoLa stands for Component Concept Lattices.

ILocZoomMap
findMap(departAddr:Location,arrivAddr:Location):ZoomableMap
extractRoadNetwork(m:Map):Hypergraph<Place,Road>
map(place:Location):ZoomableMap
RouteCalculation
IConversion
convert(addr:MailAddr):GPScoord
convert(addr:GPScoord):MailAddr
IRoute
route(departAddr:CompleteLocalization,arrivAddr:CompleteLocalization, date:Date):Route
distance(departAddr:CompleteLocalization,arrivAddr:CompleteLocalization):Float

replaces
replaces

IGpsMap
findMap(departAddr:GPScoord,arrivAddr:GPScoord):Map
extractRoadNetwork(m:Map):Hypergraph<Place,Road>
map(place:Location):Map
PubTransportRouteCalculation
IConversion
convert(addr:MailAddr):GPScoord
convert(addr:GPScoord):MailAddr
IPubTranspRoute
route(departAddr:MailAddr,arrivAddr:MailAddr):PubTranspRoute
distance(departAddr:MailAddr,arrivAddr:MailAddr):Float
duration(departAddr:MailAddr,arrivAddr:MailAddr):Float
buyTicket(ptr:PubTransRoute):TranspTicket

ILocZoomMap
findMap(departAddr:Location,arrivAddr:Location):ZoomableMap
extractRoadNetwork(m:Map):Hypergraph<Place,Road>
map(place:Location):ZoomableMap
TouristicRouteCalculation
ITouristicAgenda
agenda(departAddr:GPScoord,arrivAddr:GPScoord,date:Date):TouristicAgenda
ITouristicRoute
route(departAddr:GPScoord,arrivAddr:GPScoord,date:Date):TouristicRoute
distance(departAddr:GPScoord,arrivAddr:GPScoord):Float

replaces
replaces

ILocMap
findMap(departAddr:Location,arrivAddr:Location):Map
extractRoadNetwork(m:Map):Hypergraph<Place,Road>
MuseumRouteCalculation
IMuseumAgenda
agenda(departAddr:GPScoord,arrivAddr:GPScoord,date:Date):MuseumAgenda
exhibitions(m:Museum, date:Date):List<Exhibition>
IMuseumRoute
route(departAddr:GPScoord,arrivAddr:GPScoord,date:Date):MuseumRoute
distance(departAddr:GPScoord,arrivAddr:GPScoord):Float

ILocZoomMap
findMap(departAddr:Location,arrivAddr:Location):ZoomableMap
extractRoadNetwork(m:Map):Hypergraph<Place,Road>
map(place:Location):ZoomableMap
BotanicRouteCalculation
IBotanicGardenAgenda
agenda(departAddr:GPScoord,arrivAddr:GPScoord,date:Date):BotanicAgenda
species(botRoute:BotanicRoute, date:Date):List<BotanicSpecies>
IBotanicGardenRoute
route(departAddr:GPScoord,arrivAddr:GPScoord,date:Date):BotanicRoute
distance(departAddr:GPScoord,arrivAddr:GPScoord):Float

Figure 1. Classification of route computation components

shows the descriptions of these components (the three leaf components and the `TouristicRouteCalculation` component) along with other informations that will be explained further in the paper.

Figure 2 presents the specialization hierarchies of data types involved in component functionalities. `Map` is specialized by `ZoomableMap`. `Route` is successively specialized by `PubTranspRoute` (public transportation route), `TouristicRoute`, `BotanicRoute` which define routes for tourists interested in visiting botanic gardens and `MuseumRoute` which does the same for museums. `TouristicAgenda` is derived along botanic garden or museum interests. `Location` (just described by a name) is specialized by `GPScoord` (which contains the place's GPS coordinates), `MailAddr` (which contains the place's address) and `CompleteLocalization` which integrates all information.

The `PubTransportRouteCalculation` component provides four functionalities. The first three calculate a public transportation route, distance and duration between given departure and arrival mail addresses. The fourth allows to buy a transportation ticket that corresponds to a given route. The component also requires localization conversion functionalities and map functionalities that produce a general map that covers route departure and arrival points, a road network and a detailed map centered on a given location.

The `TouristicRouteCalculation` component calculates a touristic route that integrates scenic sites that are close to the standard route and the distance covered by such a touristic trip. It also suggests an agenda of events of interest for tourists.

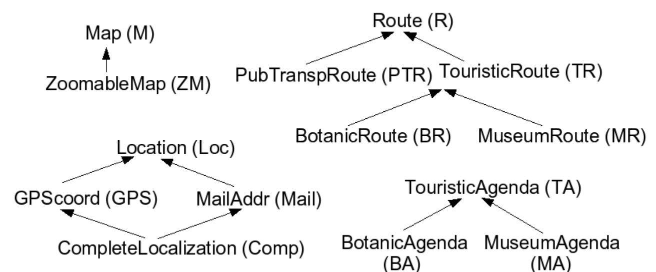

Figure 2. Type hierarchy for route computation components

The `BotanicRouteCalculation` (*resp.* `Museum-RouteCalculation`) component calculates routes (and the corresponding distances) that integrate botanic gardens (*resp.* museums) close to the standard route. It also provides an agenda of gardens (*resp.* museum) events and a functionality that lists the specific botanic species (*resp.* exhibitions) tourists may find in the visited gardens (*resp.* museums).

In the current component directory, there are many of such components with similar provided and required functionalities. If the directory has a flat structure, the user (human designer or automatic procedure) has to carefully browse the whole directory to find relevant components for connection or substitution. Even in cases where the component library can be searched through with keyword-based requests, the user still has to browse the resulting set of functionalities and has no clear information on component relevance. For these software development or maintenance tasks, it would be useful to easily know:

- which components can be assembled with a given component,
- which components behave similarly, to quickly find an approximate substitute to a given component,
- which new components should be added to the repository to improve future development and maintenance tasks.

We can answer these questions rather easily by examining the component classification we propose to build in this paper (*cf.* Figure 1). Firtsly, we provide classification links with replacement semantics. For example, `MuseumRouteCalculation` is classified as a possible substitute of `TouristicRouteCalculation`. Secondly, new (abstract) component types (*e.g.,* `RouteCalculation`) emerge: they generalize existing (concrete) components and their implementation is suggested. Indeed, defining a component assembly with such a general component rather than specialized ones would make the assembly more reusable and tunable. Before explaining the basics for substitution and component type emergence, next section gives an overall view of the approach.

3. The three-step classification process of CoCoLa

In CoCoLa, components are automatically indexed in a "yellow page"-like directory designed specifically for searching components that can assemble to or substitute to a given one. The process of component classification relies on the syntactical information provided by the external views of components (functionality signatures grouped in interfaces of either provided or required direction). It uses FCA to calculate concept lattices [2] that order this syntactical information so that searching becomes as easy as lattice traversal.

Our approach decomposes into three steps. At each step, FCA is used to build a classification ordering the artifacts according to the substitution principle.

- As a first step, classifications of provided and required signature functionalities are built using the input and output parameter types. Existing signatures are organized using the substitution order and new signatures emerge.
- As a second step, classifications of provided and required interfaces use the functionality signature classifications built at previous step. Existing interfaces are organized using the substitution order and new interfaces emerge.
- As a last step, component classification uses the classifications of interfaces. It indicates possibilities for component substitution and assembly and generates new component descriptions that provide more general designs.

These three steps are illustrated on the didactic example of Section 2. Each of them is described in one of the three following sections.

4. Classifying functionality signatures

At the first level, the substitution principle establishes that a functionality substitutes to another one if *it requires less and provides more*.
Require less means that the substitute functionality can:

- Generalize input parameter types in provided signatures (or remove input parameters),
- Generalize output parameter types in required signatures.

Provide more means that the substitute functionality can:

- Specialize output parameter types in provided signatures,
- Specialize input parameter types in required signatures (or add extra input parameters).

Let us first consider the case of the provided `route` functionality. Table 1 shows the four `route` signatures used in components of the repository.

The table encodes knowledge about these signatures. Several kinds of descriptions, based on different underlying connection and substitution models, can be used that need more or less adaptation capabilities. In this paper, we use a simple model where the order of parameters and the number of parameters of a given type do not matter. This model is very loose: It requires syntactic adaptations in Java-like languages, but stricter models could be encoded and implemented with the same construction techniques. Tables thus encode the sets of input and output parameter types. For our current application, we suppose that there is a single output parameter type (return type). Further work will generalize to multiple outputs. In tables, $\times$ is used

for a property owned by the signature, while $\otimes$ is used for an inferred property. Inferences are deduced from the substitution principle. For example, the `route` functionality signature of first row owns `Mail` as one of its input parameter types, and `PTR` as its return type. Inference on input parameter types is based on the rule which states that in the substitute, input parameters of provided functionality signatures can be generalized. In the other perspective, it means a functionality can substitute to another one the input types of which are more specialized. For example, input parameter type `Mail` in `route` signature (first row) may substitute to input parameter type `Comp` (specialization of `Mail`). Inferences on output parameter types obey the reverse rule.

To encode the removal of input parameters, we need to encode that a signature does not contain a parameter. We add $\neg p$ (also denoted by Np later in some figures) as a fictive parameter type, with the meaning that $\neg p$ substitutes to p, because a signature that does not own p substitutes to a signature that owns p. For example, the `route` functionality signature of first row owns $\neg D$ as one of its input parameter types (D is inferred), because it does not own `Date` as an input parameter type.

After the description of signatures, FCA is used to build the classification. Given a table describing a set of objects that own properties, FCA enables to calculate a lattice ordering concepts extracted from the table. A concept is a maximal subset of objects connected to a maximal subset of properties, such that all objects own all properties. For example, as all route signatures own the property set {in_Comp,in_D,out_R}, they define a concept together with this set of properties (*cf.* $Concept_0$ on Figure 3). The whole lattice is presented in Figure 3. Concepts are represented with a simplified form where inherited properties (top to bottom) and inherited objects (bottom to top) are omitted. Figure 4 gives an interpretation of the lattice by rebuilding a functionality signature for each concept of the lattice (except for the bottom one). Valid substitutions can be read bottom-up in this lattice: a functionality can replace a higher (more general) one.

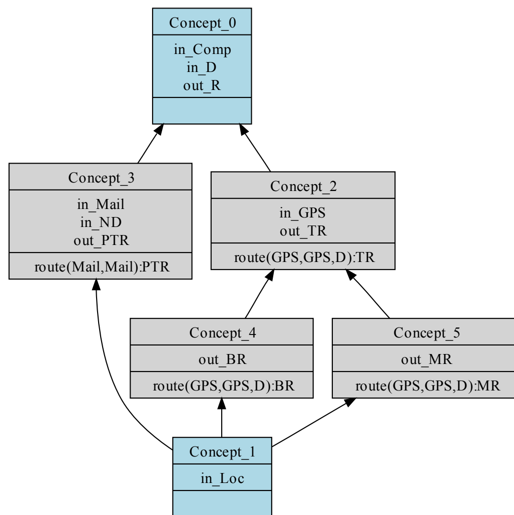

Figure 3. Lattice of route signatures

Similarly, we can produce classifications for all functionality names from the example[2]. The case of required functionality signatures is dealt with reverse encoding (as detailed in [3]).

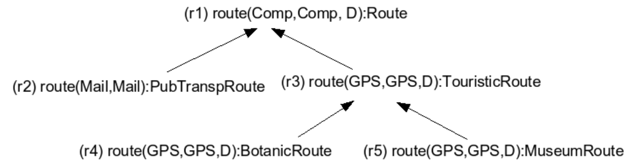

Figure 4. Classification of route signatures

5. Classifying interfaces

In this model, as in object-oriented languages and component models, interfaces are collections of functionality signatures. Encoding interfaces will thus rely on the lattice of functionality signatures and apply the substitution principle for inferences. At this level, the substitution principle states that in an interface that can substitute to another one, we can find less required functionalities and more provided functionalities. This implies two opposite encodings — one for provided interfaces and one for required ones. Provided interfaces are described in a table (Table 2 for the example) where columns correspond to relevant concepts from the functionality signature lattices. Some concepts correspond to existing functionality signatures while others have been created by factorization. This is the case of `r1=route(Comp,Comp,D):R` (interpretation of $Concept_0$ from Figure 3). A $\times$ character is used when the provided interface owns a functionality signature. An interface which owns a signature can substitute to another one, which owns a signature higher in the lattice (it can replace it). For example, `IPubTranspRoute` owns `r2` which substitutes to `r1` (*cf.* Figure 1) thus $\otimes$ is set for (`IPubTranspRoute`, `r1`). Figure 5 gives the resulting classification of provided interfaces. Substitutions can be read bottom-up. New interfaces emerge from this classification process: $Concept_1$ represents a new provided interface that factorizes `r1` (`route(Comp,Comp,D):R`) and `d1` (`distance(Comp,Comp,D):Float`). It will be denoted `IR` for `IRoute` in the following. In the case of required interfaces, the substitution principle needs adding the knowledge of which functionality signature a required interface has

2. The whole figure set of the example is available online: http://www.lirmm.fr/∼huchard/RouteComponent/

Table 1. Encoding route signatures (provided point of view). Type names are shortened

	In parameters						Out parameters				
	Loc	GPS	Mail	Comp	¬D	D	R	PTR	TR	BR	MR
route(Mail,Mail):PTR			×	⊗	×	⊗	⊗	×			
route(GPS,GPS,D):TR		×		⊗		×	⊗		×		
route(GPS,GPS,D):BR		×		⊗		×	⊗		⊗	×	
route(GPS,GPS,D):MR		×		⊗		×	⊗		⊗		×

not. Similarly to the case of required signature encoding, a required interface which has not a given functionality signature will be described as possibly substitutable to all interfaces that have any form of the functionality.

6. Classifying components

In this model, components can be seen as collections of directed interfaces, as in most component models. This description is often referred to as their external view. Ensuring component substitution demands that less interfaces are required, while more provided interfaces are possible. Following the same principles, Table 3 shows a description of the external view of components. Components are described by the interfaces they own and interfaces inferred by substitution rules. Ownership of a required (*resp.* provided) interface implies inference of required (*resp.* provided) interfaces higher in the corresponding lattice. When a component does not own a required interface (*e.g.,* `TRC` does not own `IConversion`), all the forms of this interface can be admitted for substitution and are inferred. Figure 6 presents the resulting lattice which can be used for building the classification of Figure 1. $Concept_0$ is the `RouteCalculation` component of the top of the classification. $Concept_1$ was removed for the sake of simplicity but could also be interpreted and included inside the result. $Concept_2$ would be interpreted as the component containing all interfaces and it is unlikely that designers would be interested to develop it as a new component.

7. Implementation and Experimentations

In order to implement the process presented in this paper, we first define a meta-model which groups all the concepts needed in the process. It sets a vocabulary for the component-based architecture descriptions that are used to build concept lattices. The component descriptions from the component repository, which provide the external views of components we want to classify, are first transformed into instances of this meta-model. The resulting models (now expressed in a common vocabulary) are then used to generate context tables. Next paragraphs first present this meta-model and then detail the architecture and functioning of the CoCoLa prototype tool.

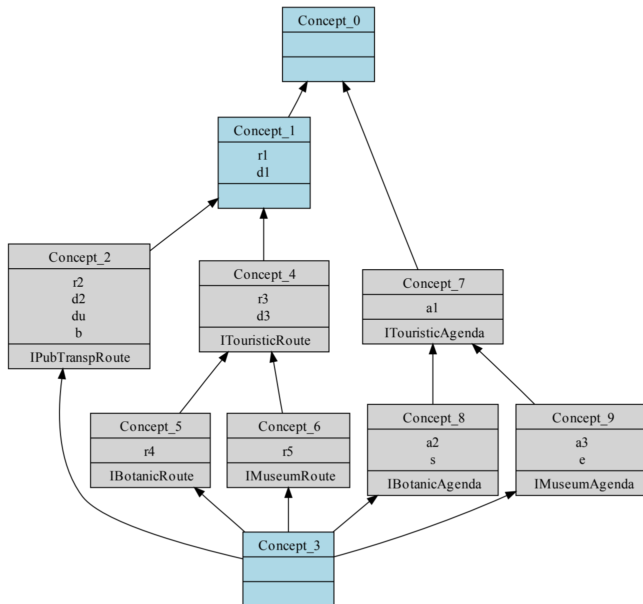

Figure 5. Classification of provided interfaces

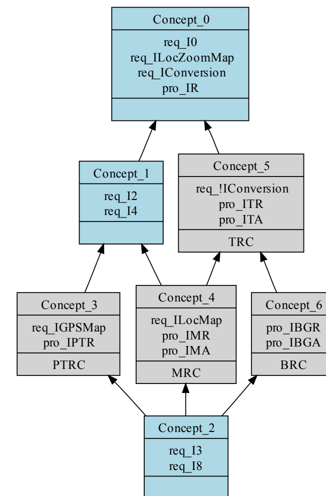

Figure 6. Classification of components

Table 2. Encoding provided interfaces

	r1	r2	r3	r4	r5	d1	d2	d3	du	b	a1	a2	a3	e	s
IPubTranspRoute	⊗	×				⊗	×		×	×					
ITouristicRoute	⊗		×			⊗		×							
IMuseumRoute	⊗		⊗		×	⊗		×							
IBotanicRoute	⊗		⊗	×		⊗		×							
ITouristicAgenda											⊗				
IMuseumAgenda											⊗		×	×	
IBotanicAgenda											⊗	×			×

Table 3. Encoding components. IR represents $Concept_1$ of provided interface lattice. Names have been shortened.

	Required interfaces										Provided interfaces							
	I0	ILocZM	I2	I4	I3	I8	IGPSM	ILocM	¬IConv	IConv	IPTR	IBGR	IMR	ITR	IR	IMA	IBGA	ITA
PTRC	⊗	⊗	⊗	⊗	⊗	⊗	×			×	×				⊗			
TRC	⊗	×							×	⊗				×	⊗			×
MRC	⊗	⊗	⊗	⊗		⊗		×	×	⊗			×	⊗	⊗	×		⊗
BRC	⊗	×							×	⊗		×		⊗	⊗		×	⊗

7.1. CoCoLa Component Meta-Model

The metamodel of Figure 7 depicts the grammar of the model we adopted for component-based software architecture descriptions in our context. In this meta-model, we consider a component as a software artefact defining a set of directed interfaces. Each interface can thus be provided or required by a component. An interface is described by a set of functions which declare parameters in input, output or the two simultaneously (inout parameters). Each function can additionally declare exceptions of a certain type. This model supports hierarchical descriptions of components. Thus, a component can have sub-components within its definition. Classes, components and interfaces have in common the meta-class `StructuredType`. As in most programming languages, this is considered as a specific kind of the concept `Type`, in the same way as for the meta-class `PrimitiveType` which conceptualizes integers, booleans, and other basic types. Contrary to primitive types, structured ones can have supertypes and known subtypes. Functions, parameters and types are considered as named elements: their name serves as an identification key in the whole set of artefacts.

7.2. Component Ordering as an Automatic Transformation Process

We developed a prototype tool called CoCoLa that implements the proposed automatic component ordering process. Figure 8 provides an overview of the CoCoLa tool architecture. It receives as input a set of component-based architecture descriptions defined in Fractal [4] ADL, the Java implementation of these components (which take the form of Java classes) and their interfaces, and a tree of type hierarchy. The `XMI-Builder` module of CoCoLa produces an XMI [5] document which merges all these (input) definitions and represents an instance of an Ecore[3] metamodel defined as a concrete implementation of the metamodel of Figure 7. The tool then starts to generate context tables for functionalities grouped by names, provided (resp. required) interfaces and components. These tables are serialized as CSV (Comma-Separated Values) files. Starting from these definitions, the tool builds the concept lattice for each context table. For doing so, the `Lattice_Builder` component of CoCoLa uses an external library, called `erca`[4] to produce DOT[5] files containing the concept lattices of each architectural artefact.

The CoCoLa tool has a graphical user interface mod-

3. http://www.eclipse.org/modeling/emf/
4. http://code.google.com/p/erca/
5. DOT is a plain text graph description language.

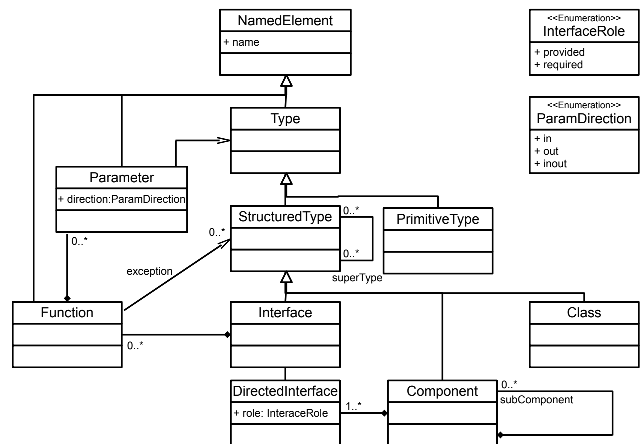

Figure 7. CoCoLa component metamodel

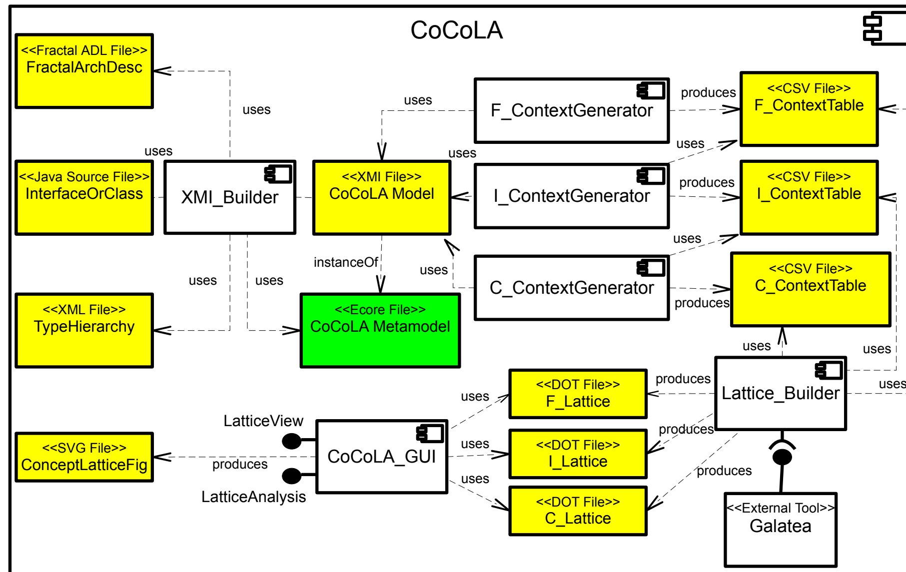

Figure 8. A simplified structural view of the architecture of CoCoLa

ule which provides two categories of functionalities. First, it builds an SVG[6] description of lattices of the different artifacts and draws their graphical representation in the GUI. Second, it implements some tasks on these lattices to simulate requests that could be directed to a component directory. For example, it lists the components that can replace and the components that can be connected to a component choosen by the user. Similar simulated user requests can be run for interfaces.

7.3. Experiments on the Dream Library

We experimented our approach on a set of Fractal components issued from Dream[7]. Dream is a component-based framework dedicated to building communication middleware. It provides, among others, a component library implementing various communication paradigms: group communications, message passing, event-reaction, publish-subscribe, etc.

We measured a set of metrics on this library and we obtained the following results:

- Total number of component definitions: 170
- Total number of interfaces: 283
- Average number of interfaces per component: 1.92
- Number of provided interfaces: 127
- Number of required interfaces: 156
- Maximal Depth of component (resp. interface) hierarchies (typing point of view): 2 (resp. 7)
- Total number of functions: 799
- Number of functions in required interfaces: 561
- Number of functions in provided interfaces: 238
- Total number of parameters: 1007
- Average number of parameters per function: 1.26

This library is of relatively average size. We nonetheless observe that it defines more than 1000 parameters that appear in almost 800 functions. All of these functions are declared in 283 interfaces provided or required by 170 component definitions. There is almost the same number of required and provided interfaces (22% of required interfaces more). The number of functions in required interfaces is however more important than in provided ones (nearly 2,4 times more) and the number of parameters of non-primitive types (arrays, classes and interfaces) represents about 94% of the total number of parameters.

It is obvious that we cannot present in this paper the generated lattices, because of their large size and the complexity to users to navigate inside. In the following paragraphs, we present however some metric values measured on these lattices and some interesting interpretations of them. We limit voluntarily our study to the provided part of the components to be more concise.

In the Dream library, there are not much variations on the signatures, and functionality signature lattices are mostly reduced to one concept.

6. SVG (Scalable Vector Graphics) is an XML-based file format for describing vector graphics.

7. ObjectWeb Dream project Website: http://dream.ow2.org/

The lattice of provided interfaces[8] contains:

- 51 concepts,
- 96 edges in transitive reduction (edges of the whole ordering represent potential substitution),
- 4 factorization concepts (they represent new provided interfaces) which factorize one signature,
- 5 merged concepts (they group interfaces with the same provided external view) which group between 1 and 5 interfaces that share 2 and 3 signatures.

Figure 9 shows an example of a factorization concept: C_{49} factorizes `bind( ): OutgoingPush` for interfaces `ChannelProtocol` and `TCPIPProtocol`.

The lattice of component external views[9] contains:

- 55 concepts,
- 101 edges in transitive reduction (edges of the whole ordering represent potential substitution),
- 7 factorization concepts (they represent new provided external views), which factorize one interface,
- 14 merged concepts; one groups 58 composite components that don't have external interface ; the other 13 concepts group an average of 4 components that share one or two interfaces.

Figure 10 shows the `TaskManager` component which is classified as a possible substitute (relatively to provided point of view) to `ActivityManager`. The `ActivityManager` and `ActivityManagerType` components share the same provided external view.

This case study shows the feasibility of the technique; Lattices of the provided part of components have reasonable size and help identifying opportunities for substitution and new definitions. They also give an overall view of the library thanks to the non-flat organization and constitute a structure suitable for navigation tools.

8. Related work

In the literature there are many works regarding the organization of software libraries and the retrieval of components. Iribarne et al. [6] define the requirements for a component trading service which enables to publish, query and retrieve existing components. Component descriptions encompass different kinds of information: functional (syntactic definition of interfaces), formal (behavior and collaboration protocol), non-functional (semantic properties) and business (company affiliation). Regarding syntactical informations, the use of exact and relaxed matching schemes, based on substitution and specialization rules on component types (sets of interfaces), is suggested. However, this work only outlines the features of such registries and is more focused on the conceptual definition of a component trading service. Contrarily, our work aims at defining practically how the content of a component registry can be built to efficiently support component indexing and retrieval.

In the code conjurer tool [7], interfaces are extracted from test-cases and used for finding classes thanks to the merobase[10] component finder. Required part is dealt through an automated dependency resolution mechanism. In our proposal, we combine provided and required aspects in classifying components and suggest the development of more generic components.

Zaremski and Wing [8] propose an extensive study of functionality signature matching rules. Relaxed matching is based on functionality substitution principles, stated as predicates on the pre- and post-conditions of the functionalities. The use of such matching rules is advocated to design search mechanisms for function libraries but no concrete solution to structure and index the content of such repositories is defined. Our component substitution rules are derived from this work, which is an extension of the common specialization rules of strongly-typed object languages [9] to support relaxed matching. Our work proposes an adaptation of this work to components, to deal with directions and the iterative classification of more complex syntactical types, from functionalities up to components.

Existing component registries, also called trading services, such as Corba Trading Object Service [10], conform to the principles of the ODP standard [11]. Component service advertisements are published to a component registry. Service types can be structured as a specialization hierarchy. As opposed to our approach, the service type hierarchy is built manually and is static [12]. Moreover, the classification of the components is also manual and explicitly defined in the component advertisement. Finally, these component registries are purely service oriented and only contain provided interface definitions. In our proposal, the content of such registries is extended to required interfaces and whole component types in order to support various architectural construction and evolution processes.

Comparable registries have been studied for web services [13]. Registries usually provide simple data models with limited capabilities for structuring their contents. UDDI and WSDA registries for instance are designed essentially to discover existing web services. Their entries do not contain any detailed descriptions but a link to comprehensive external descriptions (generally in WSDL) maintained by the web service providers. Web services are classified into business categories associated with keywords that enable their pre-selection. On the contrary, ebXML registries [14] contain extensive descriptions of web services, stored thanks to a complex, extensible data model, enabling multiple classifications. However, classification is handled manually

8. http://www.lirmm.fr/~huchard/RouteComponent/fixtest6/dot/Interface/pro-interface_treillis.dot.svg

9. http://www.lirmm.fr/~huchard/RouteComponent/fixtest6/dot/Components/component_treillis.dot.svg

10. http://merobase.com/

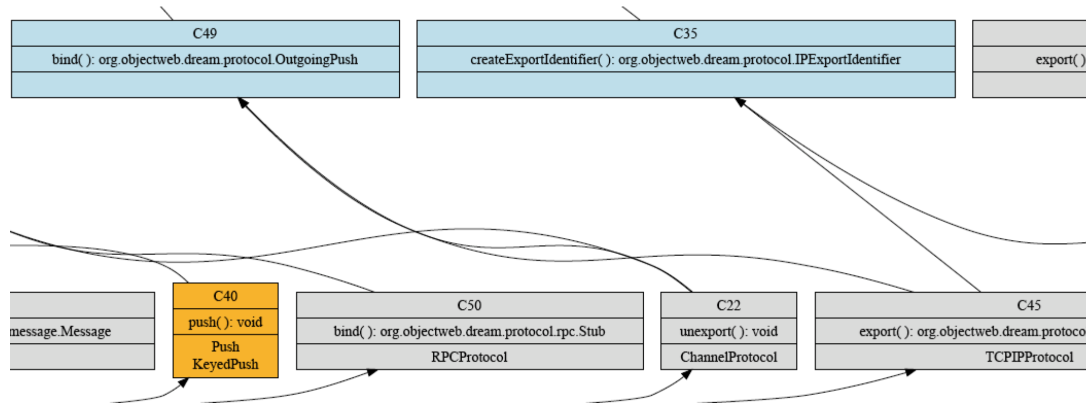

Figure 9. An excerpt of the provided interface lattice of Dream library

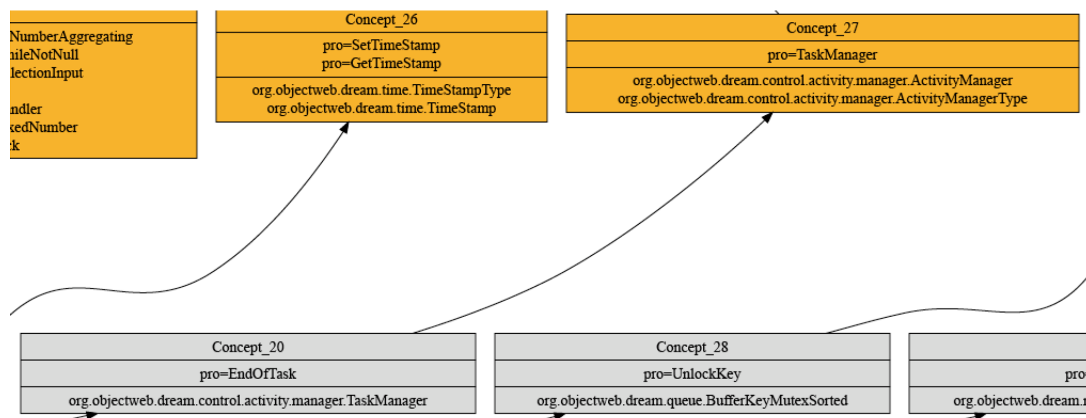

Figure 10. An excerpt of the component provided external view lattice of Dream library

by the web service providers, defining explicit classification information in their web service registration requests. As for previous service registries, this leads to poorly structured contents and erroneous retrievals.

FCA [15] has been studied as a solution to automatically organize browsable functionality libraries [16]. Queries are formulated incrementally with keywords, narrowing the number of potential results as the query becomes more precise. FCA is used to structure the set of keywords and requests are handled as traversals of the resulting concept hierarchy which does not necessarily reflect specialization relations between the syntactical types of functionality signatures. Fischer [16] uses attributes which represent fragments of the formal specifications of functionalities (simple pre- and post- conditions).

In the context of web services, machine learning techniques (clustering) are applied to the textual documentations of the services to cluster them and identify meaningful keywords [17], [18]. FCA is then used in a second step to structure the classification and drive the matching between queries and the indexed services.
As compared to this work, we not only deal with the provided services but also with the required services. Moreover, we propose a multi-level classification process to handle the various syntactical types which define components (functionalities, interfaces, whole components). In this paper, we extend the work in Arévalo et al. [3] to obtain better factorization on provided functionality signatures and to include the support of required functionality and interface removal in a substitute.

9. Conclusion

The contribution of this paper is twofold. It first presents an automated process for classifying components from their external descriptions. This process is based on type-theory (we only use syntactic information) and uses FCA to iteratively build lattices that provide functionality signature classifications, interface classifications and component classifications. Compared to our previous work, the semantics of substitution has been extended to encompass artefact addition or removal when applicable.

It then provides a description of the CoCoLa prototype tool that implements the aforementioned process. Thanks to

a pivot meta-model, component descriptions from various formats are translated into comparable models (instances of the common meta-model). These descriptions are then processed to build context tables and lattices. Experiments have been run on the Dream component library (that comes from a real-world component-based framework) and show the feasibility of our approach as it allows to identify possible component substitutions and gives readable classification of the components.

Perspectives for this work still are numerous. On the theoretical aspect, it would be interesting to run systematic comparisons on various substitution semantics (from strict typing to loose matching with more adaptations). Adding the capability of identifying variations in function names with natural language techniques would be of great interest. Adding the treatment of metadata on variants [19] and including non-functional attributes for components is also a large field to explore: non-functional attributes could allow to provide additional filtering steps to select components with more accuracy (as done in [20]). On the experimental point of view, we wish to further use CoCoLa on real component repositories to try and identify the combinatorial limit of the tool and provide solutions in the form of reasonably small directory interconnection. We also wish to further analyze component repositories in order to suggest component refactorings or extra component developments to enhance reuse capabilities.

Acknowledgements. Authors would like to thank warmly Nicolas Auboin, Olivier Bendavid, Nicolas Haderer and David Pallet who contributed in the development of the CoCoLa tool.

References

[1] N. Desnos, M. Huchard, G. Tremblay, C. Urtado, and S. Vauttier, "Search-based many-to-one component substitution," *Journ. of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 321–344, September/October 2008.

[2] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[3] G. Arevalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier, "FCA-based service classification to dynamically build efficient software component directories," *Int. Journ. of General Systems*, vol. 38, no. 4, pp. 427–453, May 2009.

[4] E. Bruneton, C. Thierry, M. Leclercq, V. Quéma, and S. Jean-Bernard, "An open component model and its support in java," in *Proc. of the ACM SIGSOFT Int. Symp. on Component-based Software Engineering (CBSE'04). LNCS 3054*, 2004, pp. 7–22.

[5] OMG, "Xml metadata interchange (xmi) v2.1.1 specification, document formal/07-12-02," http://www.omg.org/cgi-bin/apps/doc?formal/07-12-02.pdf, 2007.

[6] L. Iribarne, J. M. Troya, and A. Vallecillo, "A trading service for COTS components," *The Computer Journal*, vol. 47, no. 3, pp. 342–357, 2004.

[7] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.

[8] A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 4, pp. 333–369, 1997.

[9] G. Castagna, "Covariance and contravariance: conflict without a cause," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 3, pp. 431–447, 1995.

[10] OMG, "Trading Object Service Specification (TOSS) v1.0," 2000, http://www.omg.org/cgi-bin/doc?formal/2000-06-27.

[11] Information Technology Open Distributed Processing, "ODP Trading Function Specification ISO/IEC 13235-1:1998(E)," December 1998, http://webstore.iec.ch/preview/info_isoiec13235-1%7Bed1.0%7Den.pdf.

[12] R. Marvie, P. Merle, J.-M. Geib, and S. Leblanc, "Type-safe trading proxies using TORBA," in *Fifth Int. Symp. on Autonomous Decentralized Systems, ISADS, IEEE Computer Society*, 2001, pp. 303–310.

[13] S. Dustdar and M. Treiber, "A view based analysis on web service registries," *Distributed and Parallel Databases*, vol. 18, pp. 147–171, 2005.

[14] *ebXML Registry Services Specification (RS) v3.0*, http://www.oasis-open.org/, May 2005.

[15] C. Lindig, "Concept-based component retrieval," in *IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, J. Köhler *et al.*, Eds., 1995, pp. 21–25.

[16] B. Fischer, "Specification-based browsing of software component libraries," in *Proc. of the 13th IEEE int. conf. on Automated Software Engineering (ASE'98)*, 1998, pp. 74–83.

[17] M. Bruno, G. Canfora, M. D. Penta, and R. Scognamiglio, "An approach to support web service classification and annotation," in *Proc. of the IEEE Int. Conf. on e-Technology, e-Commerce and e-Service (EEE'05)*, 2005, pp. 138–143.

[18] M. Á. Corella and P. Castells, "Semi-automatic semantic-based web service classification," in *Business Process Management Workshops*, ser. LNCS 4103, J. Eder and S. Dustdar, Eds. Springer, 2006, pp. 459–470.

[19] M. Åkerholm, J. Fröberg, K. Sandström, and I. Crnkovic, "A model for reuse and optimization of embedded software components," in *29th IEEE Int. Conf. on Information technology Interface, (ITI 2007)*, June 2007, pp. 567 – 572.

[20] B. George, R. Fleurquin, and S. Sadou, "A component selection framework for cots libraries," in *Proceedings of the ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'08)*. Karlsruhe, Germany: LNCS 5282, Springer-Verlag, October 2008.