Building Hierarchical Component Directories

Nour Aboud, Gabriela Beatriz Arévalo, Olivier Bendavid, Jean-Rémy Falleri, Nicolas Haderer, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, Sylvain Vauttier

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Building Hierarchical Component Directories

Nour Aboud[d] Gabriela Arévalo[a] Olivier Bendavid[d]
Jean-Rémy Falleri[b] Nicolas Haderer[c] Marianne Huchard[d]
Chouki Tibermacine[d] Christelle Urtado[e] Sylvain Vauttier[e]

a. DCyT (UNQ), Buenos Aires, Argentina
b. LaBRI, CNRS and ENSEIRB, Bordeaux, France
c. LIFL/INRIA Nord Europe, University of Lille 1, France
d. LIRMM, University of Montpellier and CNRS, France
e. LGI2P, IMT Mines Ales & Montpellier University, Alès, France

Abstract Component-based development is the software paradigm focused on building applications using reusable software components. Applications are built by assembling components, where the required interfaces of a component are connected to compatible provided interfaces of other components. In order to have an effective building process, software architects need adequate component directories that both index available components and ease their search. So far, existing approaches provide only a limited structure and indexation to store/register components and, as a consequence, they propose an inadequate searching process. Even when the indexes are built as flat or hierarchical structures, the indexed elements are not the components themselves, but part of them, such as services or functions, making the search of the needed compatible components imprecise. To cope with searching and compatibility problems, the contribution of this paper is twofold: it is composed of both a refined methodology (improving a previous approach) and a tool to build a hierarchically structured component directory. The component directory solves the identified problems letting an architect find components that are compatible with a given specification (to be assembled) or components that can substitute to a given one. This directory uses Formal Concept Analysis to build component type hierarchies thanks to a three step classification process that successively classifies functionality signatures, interfaces and, at last, component types. The refinement of the methodology is based on a substitutability relationship between components, where the notion of not having a parameter or an interface of some type makes our methodology more robust when considering required interfaces. The tool made possible

Nour Aboud, Gabriela Arévalo, Olivier Bendavid, Jean-Rémy Falleri, Nicolas Haderer, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, Sylvain Vauttier. *Building Hierarchical Component Directories.*

In Journal of Object Technology, vol. 18, no. 1, 2019, pages 2:1–37. doi:10.5381/jot.2019.18.1.a2

to overcome scalability issues implementing several variations on the classification strategy. We present several experiments on three case studies to classify components from on-line open-source component repositories as a validation of our methodology.

Keywords Component-based development; Component directories; Component classification; Component substitution; Formal Concept Analysis; AOC-poset

1 Introduction

In the past two decades, component-based software development (CBSD) has emerged from existing software engineering paradigms as a challenging discipline for software engineers. Standards, such as UML [OMG07], have evolved their specification to integrate components as first-class modeling elements, while in the past they were considered as simple deployment units. The proposal of methodologies, languages and tools has thus been a major step to make this promising way of building software concrete [CSAC11, OMG07]. In this paradigm, applications are built by assembling several "prefabricated" software units, called *components*, which publish interfaces that make both the provided (served) and the required (used) services explicit.

There are two major concerns in component-based software development that motivate our work:

1. Building a component-based software architecture. The building principle of component-based software architectures is that the required interfaces of a component are connected to compatible provided interfaces of other components. Two interfaces of two components are compatible if a component's provided interface at least includes the functionalities[1] specified in the other component's required interface or more general functionalities. Finding a suitable component with adequate interfaces is an important concern to build a sound architecture.

2. Evolving an already built component-based software architecture in which a component is not operational anymore. When a component fails or disappears, searching for a possible substitute is another concern.

These two concerns can be tackled by providing an adequately structured component directory which classifies components and eases compatible or substitutable component search. Such a directory should be automatically built, in order to scale and have a hierarchical index that naturally classifies components according to their potential substitutability relationship.

It is important to mention that the component specification is provided by its type, which is defined by its (required and provided) interfaces and functionality signatures.

By classification, we mean that we need to group components that share specifications (component types) and provide an order among these groups. Such a classification eliminates redundancies between type definitions and provides an accurate indexing of components: a set of queried characteristics (interfaces or functionality signatures) is matched with a type which represents the set of all the components that hold

[1] In the rest of the paper, we will refer to the *functionality* of a component as methods and/or functions in an interface of a component.

these characteristics. This classification process cannot be handled manually when component directories grow bigger.

The contribution of this paper is a refined methodology (improving a previous work [AAF+09]) and a prototype tool named DICOSOFT that builds the hierarchical classifications based on specialization and substitution principles [Lis87] that are adapted to components. The classifications can thus be easily browsed to find a component that can replace another one (the type of the former component is a subtype of the type of the latter). Such a classification mechanism is implemented using the Formal Concept Analysis (FCA) framework [GW99] and the classification is based on information that characterizes components (functionality signatures, interface specifications and component structural descriptions). Our work assumes that components which are classified come from a single provider, because, in that case, specialization relations between types are known.

Compared to our previous work [AAF+09], this paper refines the definition of the substitutability relationship. Indeed, the notion of not having a parameter or an interface of a specific type enables our new methodology to take into account the removal of required interfaces in component when descending the substitutability hierarchy. It extends the scope of our previous approach by allowing more substitution possibilities.

The DICOSOFT prototype tool implements the described methodology. It introduces three variations of the classification: classifying all component types using their whole description; classifying the required and provided parts of component types separately; classifying subgroups of component types which share several characteristics.

The paper is structured as follows. Section 2 presents the state of the art and shows that no existing work automatically classifies components as we would like to do, with a substitutability relationship that considers the duality brought by their provided and required interfaces (descending the substitution hierarchy corresponding to providing more and requiring less). Section 3 illustrates the problem tackled in this paper by means of an example and presents the motivations of the work. Section 4 provides a high level overview of the three-step classification process in which we show how we classify functionality signatures, interfaces and component types based on substitution. Section 5 describes the prototype tool that we developed as an implementation of the proposed approach, introducing two other variations to provide complementary views on the components. It presents the results of applying the approach on three repositories, including one that contains around 220 components, and discusses the obtained results. Section 6 concludes this paper.

2 State of the Art

In existing approaches, *directories*, *repositories* and *registries* are key concepts to leverage reuse in CBSD [ITV04]. We will provide the different definitions from our viewpoint to give the context of our approach.

Component repositories provide stores where component implementations (code and configuration files) are saved to be later retrieved to build an application. Repositories are the place where the code of component types can be retrieved. The repositories provide some primitives to import the different component types.

Component registries act as locator for component instances. They index the actual available component instances within an execution environment.

Upon registries or repositories, *component directories* can be built to enable

providers to publish information on available components, and users to know which components are available. The latter ones search by querying components that match their requirements, and retrieve existing (matching) components.

From white to yellow pages: Early component registries were proposed in the context of distributed programming as companion facilities for middlewares. Most of them, such as Rmi registry [Pit01], Corba Cos [Sie00], JNDI[LS00] and Equinox extension registries [MVA10], provide white-pages in which components are indexed by symbolic names. White pages only store bindings between symbolic names and references of available components instances. No meta-information documents the registered components and thus queries are limited to direct name lookups. Content structuring is limited to the manual creation of directories (similar to file systems). Fewer registries, like the Corba Trading Object Service [OMG00], the Osgi Service registry [All06] or Jini [Edw00], provide yellow-pages in which component instances are indexed by the kinds of the services they provide. These latter directories conform to the principles of the Odp standard [ISO98] and are suited to component-based and service-oriented architectures [HTL+08, EH07]. Basic queries enable users to retrieve a set of components providing a service type (exact matching).

Marvie et. al.[MMGL01] propose an extension that supports relaxed matching between the queried service types and the service types provided by the indexed components. Thus, retrieved components may not exactly match the queried service types but still be compatible. Unfortunately, the service type hierarchy is not built nor maintained automatically: it is an *ad-hoc* structure, built by the component providers, using explicit specialization declarations in the component advertisements they publish.

Compared to these works, our approach is a yellow pages directory. In addition, so far, the difference is that no current component-based development framework proposes an automatic structuring mechanism of its components directories as in our approach.

Theoretical work on component specialization: Zaremski *et al.* [ZW97] extensively studied specialization rules to structure object class libraries as type hierarchies, regarding different substitution principles, as introduced in statically-typed object-oriented languages [Lis87, LW94, Cas95]. These theoretical works do not propose any practical scheme to automatically build type hierarchies. Besides, they study object type hierarchies, which can be regarded, when dealing with component definitions, as provided interface type hierarchies (no required part).

Liskov's substitution principle [LW94] is one of the founding works on type compatibility. It introduces the definition of substitution as the possibility to replace a given object by another one while guaranteeing the same behavior. One of the mechanisms that can be used to implement sub-typing is inheritance combined with sound specialization. Components need similar specialization rules (covariance for what is provided, contravariance for what is required), but adapted and extended so as to take into account not only their required interfaces (which define different semantics as compared to their provided ones) but also more complex types (component types as higher level concepts holding interfaces with opposite directions). Research work on component-based approaches proposes type concepts and specialization rules that apply to components, interfaces or services [SR06, Fis98, Lin95, GFS08]. Compared to our approach, no work studies the automatic construction of component type hierarchies nor proposes concrete directory indexing mechanisms that could be implemented and used in component registries or repositories.

Code repositories indexed with keywords or full text analysis: Most directories do not use classification and hierarchies. Code search engines, for instance Koders [2], Google code search [3], Krugle[4] and Merobase[5], are derived from web search engines and use full-text indexation to build code repositories that can be queried by keywords. Merobase [HJA08] uses meta-information (syntactic structure of classes) and not only source code (considered as raw textual documents) to support more relevant queries: keywords can be associated with syntactic elements (classes, methods, variables, etc.) to query classes with specific features. This is thus the only code search engine which can be compared to our work, as it enables to execute type-oriented queries on functions or classes. However, it is nonetheless based on keywords and full-text indexation. Thus, it only supports exact type matching (no type hierarchies) and is dedicated to object-oriented code, thus not taking into account the explicit description of a required part.

Some work has used FCA [Lin95] to structure such keyword-based indexes in order to build browsable software libraries [Fis98, SR06] using a concept lattice. In this context, a query is formulated incrementally as a set of keywords that gradually narrows a set of matching functions, as a traversal from the concept lattice root down to the concept which corresponds to the searched keywords. The concept lattice can even suggest new keywords that can be added to refine the query (keywords associated to concepts in the sub-lattice of the current concept). In the current paper, we use FCA with a very different point of view: We build several lattices for operations, interfaces, components description levels. Besides we consider provided but also required operations and interfaces.

Many kinds of meta-information can be used to index software libraries. For instance, Fischer *et al.* [Fis98] use fragments of the formal specifications of functions (elementary pre- and post- conditions) to index them by the definition of their expected effects. On the other hand, Sigonneau *et al.* [SR06] build a function index based on the syntactical types of their input and output parameters, applying covariant and contravariant specialization rules. As compared with our proposal, these works aim at building browsable functionality directories, using a concept lattice calculated in a single step process. We have introduced an original multi-step process to compute lattices for higher-level, more complex structures (component types). This enables to provide repositories that not only deal with the provided interfaces of components but also with their required ones. This is mandatory to soundly handle queries for connectable and substitutable components [DHT+08], as a tool to support architecture building and evolution processes [ZUV10].

Web service classification: In our previous work we have used FCA and RCA (Relational Concept Analysis [HHNV13]) to classify libraries of Web service interface descriptions. In [AHT+08], we have used the operation signatures in service interfaces in order to classify services. This classification does not take into account type variance in the signatures (types of input and output parameters). Matching between a signature requested by a user and the ones in the classification is exact (exact names and types). In [AHM+11], we have classified web service interface descriptions by measuring the similarity between operation signatures. This similarity is measured using semantic and lexical metrics, like *Levenshtein* distance. Then, a threshold is used in order to build binary contexts. The resulting classification enables us to

[2] http://code.ohloh.net/
[3] http://code.google.com/p/codesearch/
[4] http://krugle.com/
[5] merobase.com/

find similar operations for a failing operation invoked in a composite Web service. There are some other similar works in the field of Web service computing. Peng *et al.* [PHWZ05] propose a classification based on FCA where in the contexts, objects are Web services and attributes are their operations. In [ABC+06], the authors present different configurations of contexts in order to build classifications of Web services, based on keywords extracted from the documentation and from the input and output parameters and their simple/complex types. Other related works using FCA and RCA to classify Web services by integrating QoS includes [DMJ+10, CLL+10]. In [ADH+11], we have extended the approach proposed in our previous work to include the quality of service (QoS) and composability levels between services. The obtained classification enables us to find a set of compositions of Web services that answer some user requirements expressed as a task workflow, each task being described with a set of keywords (representing names of services, operations and their input and output parameters). In this work on Web services, we also consider QoS and composition requirements expressed as minimal accepted values in a *Likert* scale. Contrarily to the work presented in this article, work on Web services hardly considers type variance and substitution. In addition, services in the libraries are filtered according to the user requirements, so that classifications are not composed of the whole collection of Web services. This supposes that user requirements are known *a priori*. In the current classification of software components, we suppose that user requirements are not known and we do not consider QoS (which dynamically varies). Classifications are thus built only once. Besides, the current paper considers a description with required interfaces (and operations) in addition to the provided interfaces (and operations), while services expose only provided operations.

Test-based classification: In his dissertation [Hum08], O. Hummel addresses the discovery of components (source code, executable, etc.). He goes further keyword-based searching (with general search engines) and regular expressions-based searching (like in Google Codesearch). He proposes syntactical queries based on different information including name and method signature. The approach is completed by steps where (1) it is checked whether the component can be compiled, and (2) it passes tests that define the desired semantics. Our context is not using the whole Web as a software repository as it is proposed in [HA06]. In our case, we hypothesize a knowledge of a type hierarchy which is relevant to apply the Liskov approach, and a reduced vocabulary. There is also a certain normalization of the form of the signature. This is the case in the context of a specific component library, or if a uniform description has been proposed. Our purpose in this paper is not to address neither linguistic aspects, nor behavior, nor adaptation of input/output. The syntactic substitution verification that we propose is a needed step before checking other aspects as possible behavioral substitution between two components. Our contribution involves classification through an index we find meaningful regarding subtyping and navigation between existing components. Our approach can be applied to indexing a group of close components that have been retrieved with other approaches (including test-based) and described appropriately. It is useful for improving next retrieval operations, for example if a component is missing, to find a similar one. Also we consider components that are not source code components, but described as in CBSE approaches by required and provided interfaces, containing operations described by their signatures which is specific to our problem. Libraries such as Fractal libraries are described under this form and do not need additional effort to be used.

OWL-based approaches: Ontologies are used in [YRZM10] to represent concepts

of the domain and types of components and their relationships. A query is then defined as a pattern graph which is applied to component library description in order to retrieve components. Components are described by meta-data in the form of an OWL ontology that represents the domain in [GK13]. In these works, the kind of components that are dealt with is not specified. There is no syntactic description or organization (based on required / provided functionalities and interfaces) and no classification of the components is proposed. An ontology is used in [SST10] to describe many different features of components in a CBSE context, including names of required and provided interfaces, operations, parameters, programming languages, code complexity, Qos properties, etc. Ontology concept similarity is used to map a query and component specifications. But no classification involving a subtyping or a specialization ordering is proposed. In the context of Web services, authors of [BJAR11] annotate the inputs and outputs of the provided operations. They use this annotation to build a classification of services based on covariant signatures. The classification is restricted to the specific domain of geology, reducing possible problems of vocabulary and heterogeneity of description. They do not consider nor subtyping, neither the required point of view. They have only two levels of description (services and operations) which simplify the process and the classification is not built in a systematic way, as it is the case with Formal Concept Analysis approach and they do not approach the problems of component replacement. The classification is mainly designed for experts that have to choose a component to be composed in a workflow.

3 Motivation and Illustrative Example

To motivate our approach, we use the example of a component repository that contains six components, which implement various route calculation algorithms and drivers for phones and DVD players. Firstly, we illustrate how we build a hierarchical component directory and then, with a variation of the example, we show how we ease component connection and component substitution.

We work on a simple component model based on generally accepted definitions. *Components* are black boxes that embody reusable code, which is externally described by *interfaces*. *Provided interfaces* describe the component's server capabilities (provided to others), and *required interfaces* describe the component's needs as a client. To assemble components into the architecture of an application, it is needed that required interfaces connect to provided interfaces of compatible types. When we further look inside interfaces as abstract types, we see that interfaces are sets of functionality signatures. Each signature is defined with a functionality name and the input and output parameters and types. In our approach, interface type compatibility is based on their functionality signature compatibility. We must remark that the components in our approach are stateless.

Figures 1 to 3 show the components of our example and illustrate all these notions. For example, the `PubTransportRouteCalculation` component (Figure 1) declares three required interfaces (`IGpsMap`, `IConversion` and `IDVD`) and one provided interface (`IPubTranspRoute`). These interfaces group functionalities, which have typed input and output parameters. For example, the `IConversion` required interface groups three `convert` functionalities. The first `convert` functionality has an input parameter, called `addr`, of `MailAddr` type and an output parameter of `GPScoord` type. Similarly, the other components can be described.

In the following, we explain our process to classify functionality signatures, in-

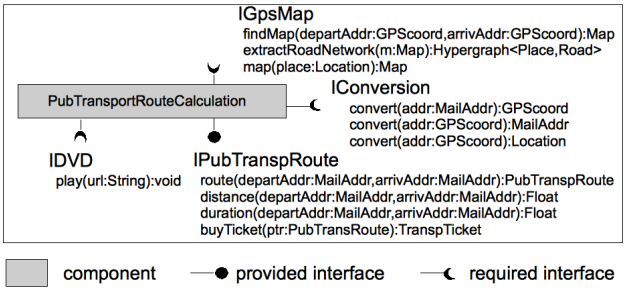

Figure 1 – A component for public transportation route calculation

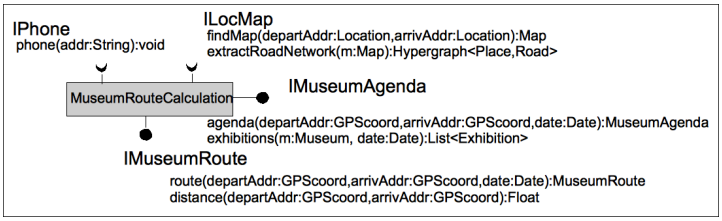

Figure 2 – A component for museum route calculation

terfaces and components. A description of the provided facet (provided interface, provided functionality signatures) in the classification steps can be found in Aboud *et al.* [AAF+09]. In the current paper, we refine the substitutability relationship for the required facet.

3.1 Building a Hierarchical Component Directory

Our three-step process relies on a specialization hierarchy of the data types[6] (Figure 4) involved in the functionality signatures of the sample components.

For example, `Route` is specialized by both `PubTranspRoute` (public transportation route) and `TouristicRoute`.

The initial flat (non hierarchical) set of components is easy to use for a given specified purpose: if a software architect wants to build a component assembly dedicated to touristic routes, she/he will use the `TouristicRouteCalculation` component (Figure 5) and connect it to a (required) component providing zoomable maps and to

[6]For clarity's sake, types that have no supertype nor subtype (*e.g.,* `Date`) are not shown in Figure 4. Complex types (such as collections or graphs) are considered to have no supertypes nor subtypes.

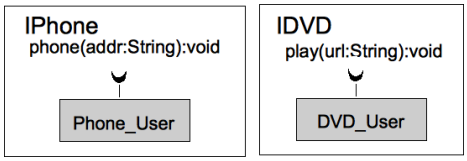

Figure 3 – Two components for information exchange

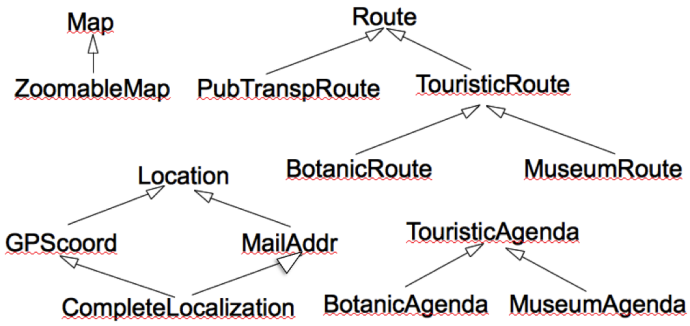

Figure 4 – Type hierarchy for route computation components' functionality parameters

components requiring the route or the agenda. However, many questions can appear when using such a flat set of components.

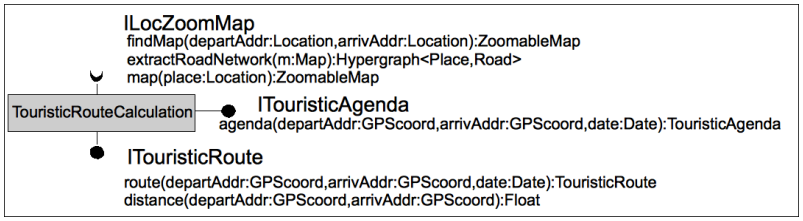

Figure 5 – A component for touristic route calculation

What can be done if the `TouristicRouteCalculation` component fails? How can the architect rapidly determine which other component can replace it? How can architects build high-level assemblies with components that are general enough to be easy to replace rather than too specific assemblies that are difficult to repair at runtime?

A suitable solution is a browsable component repository with the following characteristics:

1. Easy (non time consuming) search process. This search can be semi-automatic or automatic.

2. Relevance in finding components based on a specification defined by the developer/architect. We mean specification by the provided & required interfaces and the signatures of the methods/operations in the components. The components resulting from a query might need to be assembled at runtime and easily adapted (if needed).

3. Suggestion of new component specifications to the architects. If such new components are developed, this will increase the number of reuse possibilities.

4. Easy browsing GUIs to present a relevant subset of the components, or a point of view (required or provided) on them.

Thus, we propose a hierarchical classification, in a directory organization shown in Figure 6 for our small example. In this organization, arrows represent potential substitutability relationships. For example, `BotanicRouteCalculation` (Figure 7)

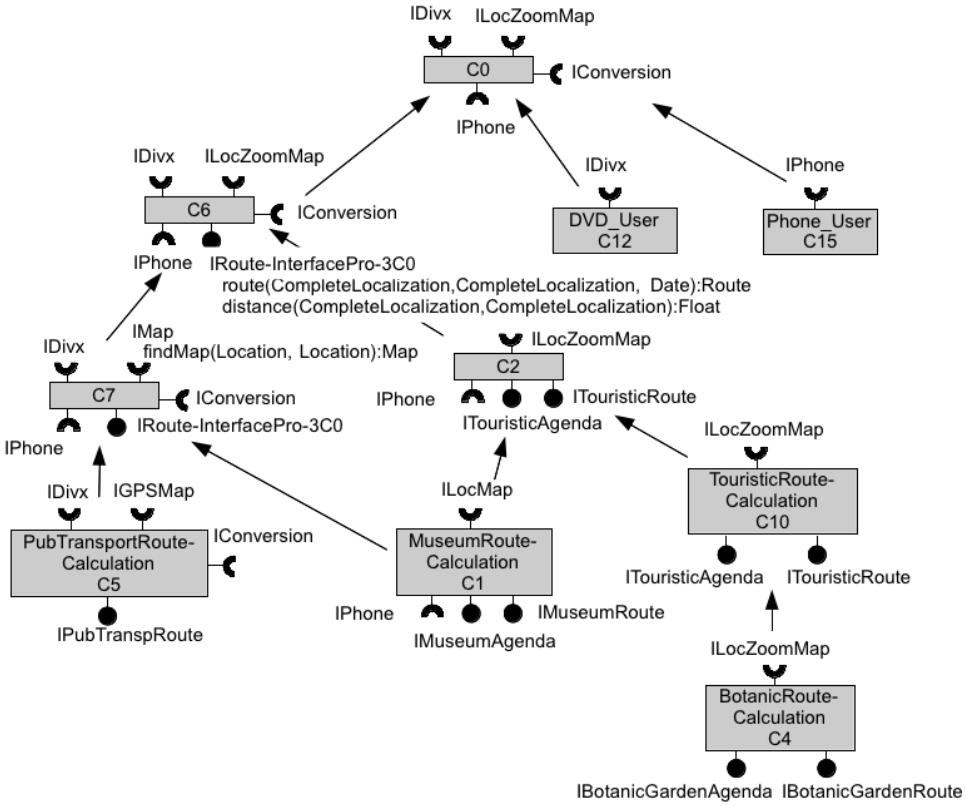

Figure 6 – Components ordered by substitutability. Arrows represent the relation "can replace" (or "can substitute itself for")

can replace a `TouristicRouteCalculation` (Figure 5). Conversely, `MuseumRouteCalculation` (Figure 2) cannot soundly replace `TouristicRouteCalculation` (Figure 5), because the latter one indeed has not an `IPhone` required interface while the former component needs to connect that interface to work. To clarify how to read the rest of the lattices in the paper, it is worthwhile to mention that the C_x notation (where x can be between 1 to 10) represents concepts, and that if a concept corresponds to a component, the name of the component is inside the box representing the concept, above the name of the corresponding concept. For example, we have the name of *TouristicRouteCalculation* above C_{10}, meaning that the concept C_{10} defines the component *TouristicRouteCalculation*.

Besides the concrete aforementioned components, the unnamed ones (defined by C_0, C_2, C_6, C_7) correspond to new abstract suggested components. Their insertion inside an assembly would increase replacement possibilities and produce a more reusable assembly. For example, if we chose to implement the component defined by C_2, an assembly using it (instead of the `TouristicRouteCalculation` component, for example) would be more reusable and easier to repair. Indeed, in case that component defined by C_2 failed, any of the three touristic components (that are lower than one defined by C_2 in the classification) could be used to replace it. These new, more general, components could also be used in abstract assemblies, or architectural patterns.

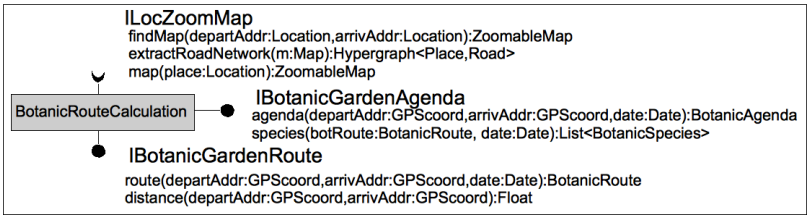

Figure 7 – A component for botanic route calculation

3.2 A Component Substitution Scenario

Figure 8 shows two components to be used in a substitution scenario (shown in Figures 9 and 10) that uses a variant of our example with the `TouristicRouteCalculationForPhoneDevice` and `SimpleTouristicRouteCalculation` components that both have a different number and different types of interfaces.

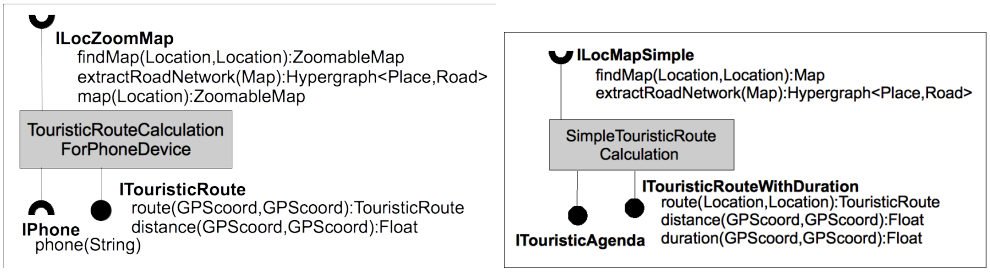

Figure 8 – `TouristicRouteCalculationForPhoneDevice` and `SimpleTouristicRouteCalculation`

Figure 9 shows the initial assembly where the objective is to have the `TravelPlanner` component work. The required `ITouristicRoute` interface of the `TravelPlanner` component is satisfied by the same provided interface of `TouristicRouteCalculationForPhoneDevice` component. In turn, the `TouristicRouteCalculationForPhoneDevice` component satisfies its `IPhone` and `ILocZoomMap` interfaces connecting them to a `PhoneDevice` and a `ZoomableMapProvider` (map-delivery specialized in zoomable maps) components respectively. In this assembly, interfaces and functionality signatures all match exactly.

Now we have the problem that the `TouristicRouteCalculationForPhoneDevice` component fails. Figure 10 gives the example of a repaired assembly where a `SimpleTouristicRouteCalculation` component replaces the defective `TouristicRouteCalculationForPhoneDevice` component. In this repaired assembly, connected interfaces and functionality signatures are not strictly identical. Changes are underlined on Figure 10. This substitution can be considered as safe:

- The `SimpleTouristicRouteCalculation` component provides the additional `ITouristicAgenda` interface which can simply be ignored.
- The `SimpleTouristicRouteCalculation` component does not require an `IPhone` interface. The `PhoneDevice` component can then disappear from the assembly which continues to satisfy the requirements of the `TravelPlanner` component.

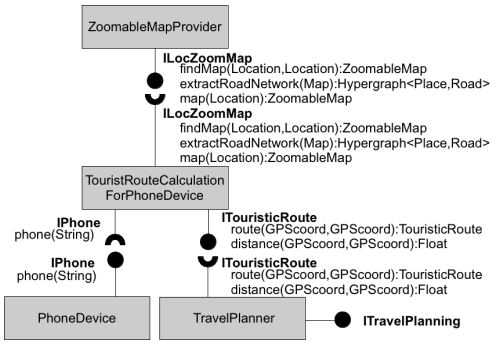

Figure 9 – An assembly

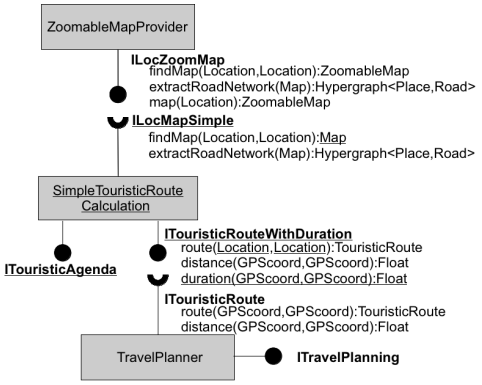

Figure 10 – A possible substitution (changes are underlined)

- The `ILocMapSimple` required interface can be connected to the `ILocZoomMap` provided interface: not requiring the `map(Location): ZoomableMap` functionality does not cause a problem because the `SimpleTouristicRouteCalculation` component can simply ignore that this functionality is available.
- When calling the `findMap` functionality from the `SimpleTouristicRouteCalculation` component, a return value of type `Map` is expected. The corresponding functionality provided by the `ZoomableMapProvider` component provides a `ZoomableMap`, which is a subtype of `Map` (it provides more), according to the type hierarchy. These return parameter types can therefore be considered as compatible, so do the functionalities and the interfaces.
- The `ITouristicRouteWithDuration` provided interface offers an additional, unused `duration` functionality. The `TravelPlanner` component can ignore it.
- There is also a variation on the input parameters of the `route` functionality. The `TravelPlanner` component assigns a value to these input parameters (of `GPScoord` type) when calling the `route` functionality. These parameter values are passed to the `SimpleTouristicRouteCalculation` component which expects parameters of `Location` type. As `GPScoord` is a subtype of `Location` (it provides more), the `route` functionality in `SimpleTouristicRouteCalculation` can be accepted in this assembly.

More complex substitution scenarios could be proposed[DHT+08]. Based on our example, we see that any scenario searching for components that meet some specification (whether it is for connection or substitution), strongly needs an adequately structured component directory that eases component search.

4 The Three-step Classification Process

In this section we provide an overview of the three-step classification process in which we show how we classify functionality signatures, interfaces and components.

4.1 A Classification Strategy based on Component Substitution

As a solution to our component search problem, this paper proposes a three-step classification process (*cf.* Figure 11) by using the type hierarchies (*cf.* Figure 4) as the low level classification. The process is described as follows:

1. Functionality signatures are classified using the type hierarchy of their input and output parameters.
2. Interfaces are classified using the classification of their functionality signatures.
3. Components are classified using the classification of their provided and required interfaces.

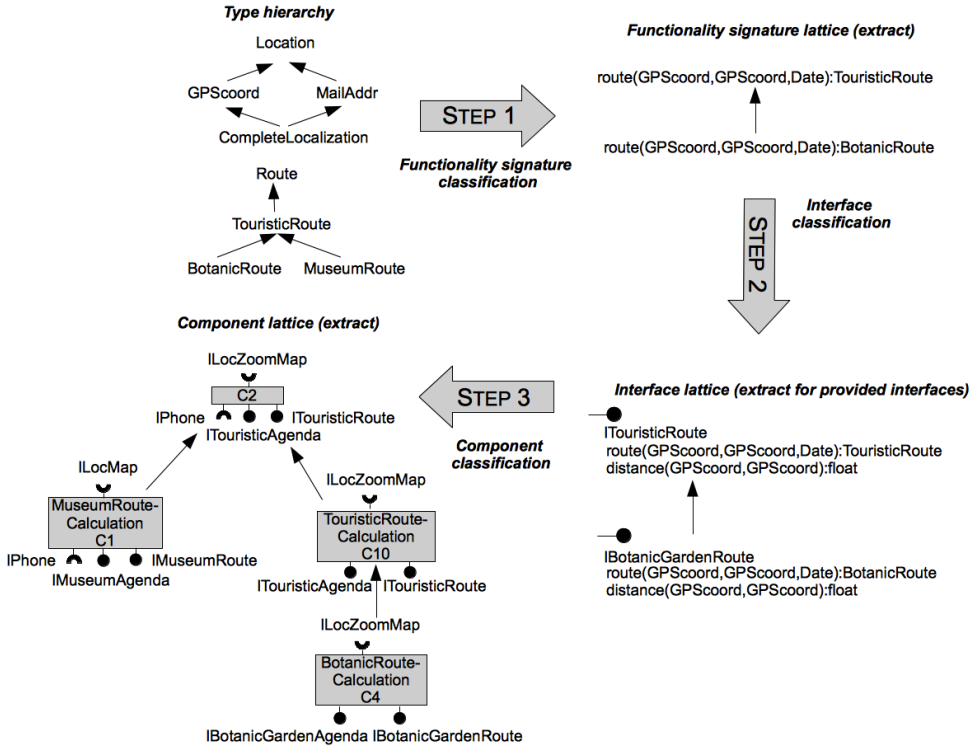

Figure 11 – Synopsis of the three-step component classification process

4.2 Classification Technique

In our approach, the classification process relies on the definition of sound component substitution. This definition is obtained by first considering functionality and interface substitution.

As a classification technique, we are using Formal Concept Analysis [Wil82] (FCA) which is a theoretical framework to analyse data. It extracts a partially ordered set (poset) of concepts (the concept lattice) from a dataset composed of objects described by attributes (the formal context). A concept is composed of two sets: an object set called the concept's extent and an attribute set called the concept's intent. These

sets satisfy the following property: the extent is the maximal set of objects that share all the attributes of the intent. Reciprocally, the intent is the maximal attribute set that is shared by all the objects of the extent. The partial order on concepts is based on top-down inclusion of intents (and, conversely, bottom-up inclusion of extents). For simplicity's sake, the concept lattice is often represented considering simplified intents (respectively simplified extents) which omit the top-down inherited attributes (respectively bottom-up inherited objects). In the figures representing concept lattices, concepts are represented as boxes where the simplified intent is the upper part and the simplified extent is the lower part.

We also consider the AOC-poset (for Attribute-Object-Concept poset [BGH$^+$14]), which is the sub-order of the concept lattice restricted to object-concepts and attribute-concepts (the poset restricted to concepts which have either a non-empty simplified extent or a non-empty simplified intent). AOC-posets scale much better than lattices as their number of concepts is bounded by the sum of the number of attributes and of the number of objects whereas the number of concepts in lattices can be exponential with regard to the minimum cardinal of the object set or the attribute set. Furthermore, the AOC-poset contains all the needed information to build the lattice, so that no data is lost.

4.3 A Simple Method for Classifying Component Types

Software component connection and substitution are our main concerns. The principles that govern component substitution thus guide our classification strategy. The general substitution principle states that a component Cp_1 can be a substitute for (can replace) a component Cp_2 if Cp_1 provides more and requires less from the environment than Cp_2. This principle has counterparts at the interface and functionality levels.

4.3.1 Classifying Functionality Signatures

A functionality can replace (be a substitute for) another functionality if it requires less from and provides more to the environment. We can intuitively consider that a required functionality behaves like a function call inside the component and that a provided functionality behaves like a function declaration in object-oriented software. For substitutability foundation, we also base our approach on a classical strongly-typed object-oriented background [Car84, Lis87, LW94] where subtypes extend their supertypes and contain more information.
A functionality can replace another if it requires less. This means:

- **contravariant input on provided side.** As input parameters in provided functionalities come from the environment, input parameter types might be generalized (less information is embedded) in provided signatures of the substitute, or might be removed.

- **contravariant output on required side.** As output (return) parameter types of required functionalities come from the environment, output (return) parameter types might be generalized (less information) in required signatures of the substitute or removed.

On the contrary, a functionality can replace another if it provides more. This means:

- **covariant output on provided side.** As output (return) parameter types in provided functionalities come from the component, output parameter types might

be specialized (more information) in provided functionalities of the substitute or output parameters might be added.

- **covariant input on required side.** As input parameter types in required functionalities come from the component, input parameter types might be specialized (more information) in required functionalities of the substitute or input parameters might be added.

We do not detail here how to encode this information in contexts for substitutability-based classification. A tricky part is that we need to encode the fact of not having a specific type as an input or output parameter. For example, the signature `route (MailAddr, MailAdddr): PubTransRoute` does not have an input parameter of `Date` type (that other route signatures have), which is denoted by `IN=!Date` in the lattice representation (intents). For further details, the interested reader can refer to the paper of Aboud *et al.* [AAF+09]. In previous papers, we have only explained the classification of provided functionality signatures, but as there are some differences with required ones, we explain the classification of functionality signatures of both directions. Functionality signatures with the same name and direction (provided or required) are classified in the same lattice.

Classifying provided functionality signatures. Figure 12 shows the concept lattice $\mathscr{L}_{route}$ of the provided `route` signatures. When an initial signature appears in the lower part (simplified extent) of a concept, this signature is exactly described by this concept. Otherwise, there is a new inferred signature that emerges from the classification process. We associate a *canonical functionality signature* with each concept. In the provided case, this canonical functionality signature is computed by removing from input and output parameters those that can replace others, that is: (1) the most general type among comparable input parameter types from the concept intent, (2) the most specific type among comparable output parameter types from the concept intent, (3) removing a type T when `IN=!`T is in the intent. For example:

- The input parameters `IN=CompleteLocalization` and `IN=Date`, and output parameter `OUT=Route` define the canonical functionality signature of concept C_0, which is a new signature.
- The canonical functionality signature of concept C_5 is defined by:
  - `IN=MailAddr` which can replace the specialization `IN=CompleteLocalization`,
  - `IN=!Date` which can replace `IN=Date`.
  - `OUT=PubTranspRoute` which can replace (thus hides in some sense) the generalization `OUT=Route`.

With the above defined substitution rules, this canonical signature does not take into account the number of parameters of a specific type. Indeed, it is considered that if a provided functionality which deals with addresses needs two data, it may also work (maybe in a downgraded form) if the counterpart provides only one, or even provides three. As for the other choices made to model substitutability, this does not diminish the generality of our classification process as changing this would simply require another encoding schema. However, we made this choice to ensure flexibility in the substitution possibilities.

Analyzing the lattice, we then learn that (for the provided functionalities):

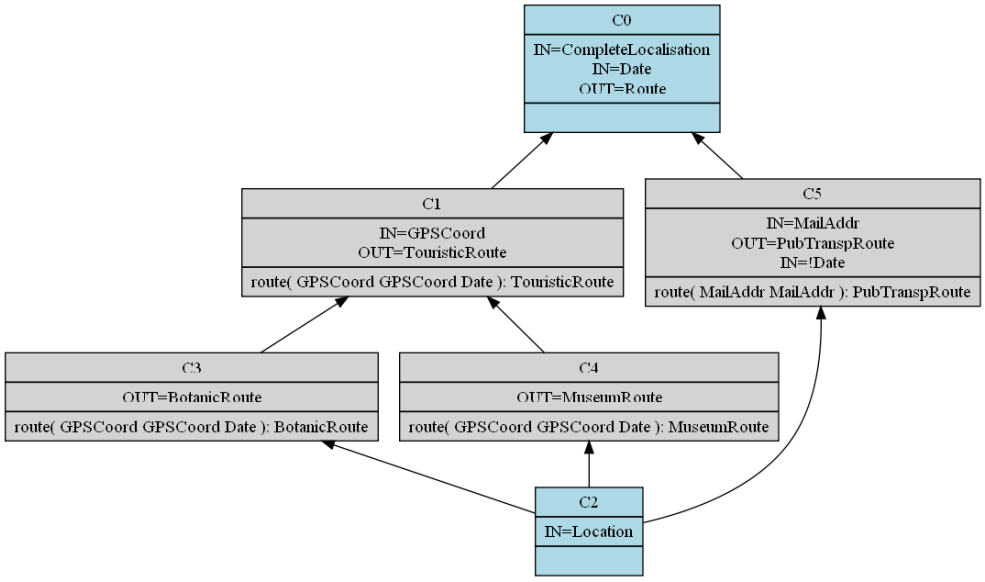

Figure 12 – The $\mathscr{L}_{route}$ lattice of provided `route` operations

- `route(GPSCoord, GPSCoord, Date):BotanicRoute` (concept C_3) can replace `route(GPSCoord, GPSCoord, Date):TouristicRoute` (superconcept C_1)
- `route(GPSCoord, GPSCoord, Date):MuseumRoute` (concept C_4) can replace `route(GPSCoord, GPSCoord, Date):TouristicRoute` (superconcept C_1)
- All known signatures (concepts C_1, C_3, C_4, C_5) can replace a new suggested signature (the canonical functionality signature of concept C_0) `route(Complete-Localisation, Date): Route`. New signatures will be denoted later with the LATTICE_ prefix. For the previous canonical signature example, and with an interpolation specific to our example (implemented in our tool), using the fact that all route signatures include two addresses, we will use the notation LATTICE_`route-Provided(CompleteLocalisation CompleteLocalisation Date):Route`
- When the extent of the bottom node of the lattice is empty, it is not considered pertinent in the classification.

In our approach, the functionality signature lattices are especially helpful to expose partial matching between signatures (with missing, additional or generalized/specialized parameters depending the direction). For example, in the lattice of provided route operations (Fig. 12), Concept C_0 presents a partial signature with two IN parameters (of types CompleteLocalization, Date) and an OUT parameter (of type Route). This signature shows what is the partial matching between the four signatures that are organized in this lattice.

Classifying required functionality signatures. The `findMap` signatures illustrate how required functionality signatures are symmetrically analyzed and classified. In required signatures, input parameter types can be specialized in a substitute. As

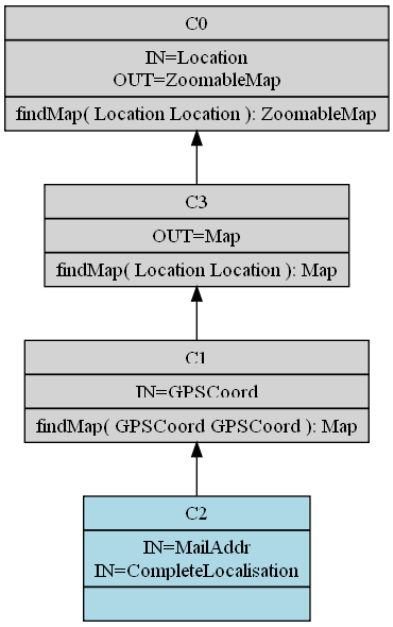

Figure 13 – The lattice of required `findMap` functionality signatures

a consequence, a signature that has `IN=GPSCoord` can replace a signature that has `IN=Location`, which refers to a supertype. Conversely, an output parameter type can be generalized in a substitute, explaining that having `OUT=Map` can replace `OUT=ZoomableMap` (refers to a subtype). Figure 13 shows the concept lattice of the required `findMap` functionality signatures. We symmetrically define the canonical required functionality signature of a concept. Analyzing the lattice, we learn that:

- `findMap(GPSCoord GPSCoord):Map` (concept C_1) is a potential substitute for `findMap(Location Location):Map` (concept C_3) and `findMap(Location Location):ZoomableMap` (concept C_0),
- There is no new required functionality signature in this example, just a substitutability-based organization of existing required signatures.

At the end of this phase, we have contexts and lattices for all functionality signatures from each provided and required facets. In our example, the 13 functionalities are organized into 26 lattices, because for each functionality we have a lattice to classify the required functionality signatures and another one to classify the provided ones.

4.3.2 Classifying Interfaces

Interfaces are classified using the functionality signature classifications. Provided and required interfaces are separately dealt with.

Classifying provided interfaces. For provided interfaces, an interface I_1 can replace I_2 if it provides more. I_1 should provide the same or more functionality signatures than I_2, or functionality signatures that can replace those of I_2 (contravariant input, covariant output).

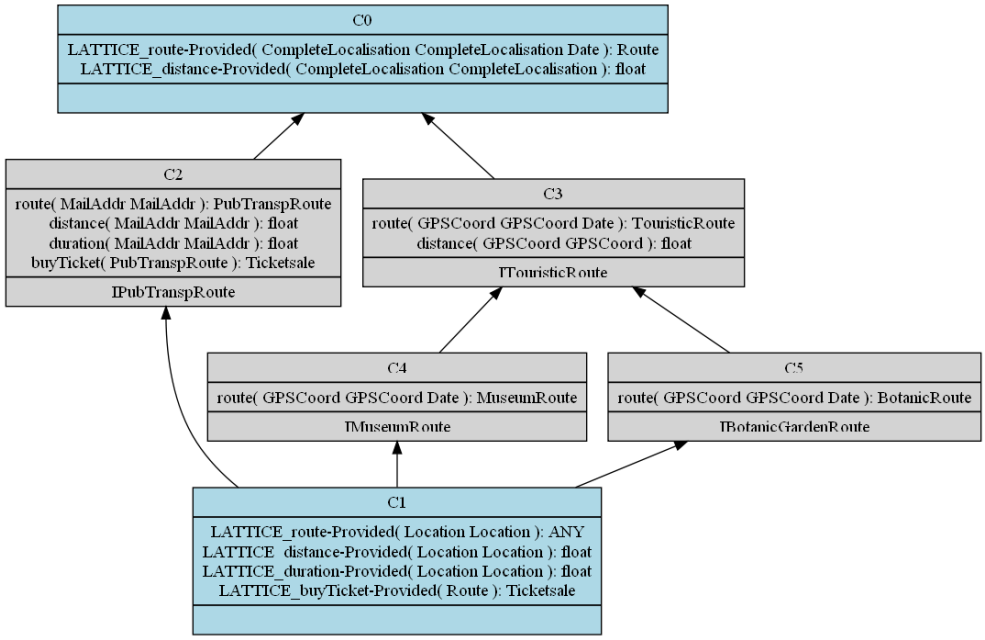

Figure 14 – The $\mathscr{L}_{routeInterface}$ lattice of provided interfaces about routes

When concepts have an empty simplified intent, we compute canonical provided interfaces. In the provided case, it means extracting the most general functionality signatures from the intent of the concept. The lattice (Figure 14) indicates that:

- `IMuseumRoute` and `IBotanicGardenRoute` can replace `ITouristicRoute`.
- A new interface (Concept C_0), whose canonical description includes `route(CompleteLocalisation,CompleteLocalisation, Date): Route` and `distance-(CompleteLocalisation, CompleteLocalisation): float` is introduced, which can be replaced by all the classified interfaces. New interfaces will be denoted later with the `Lattice_` prefix. For example this one will be the concept C_0 in *route* interface lattice, and will be denoted by `Lattice_InterfacePro-3C0`[7].

Classifying required interfaces. For required interfaces, an interface I_1 can replace I_2 if it requires less. I_1 should require the same or less functionality signatures than I_2 or functionality signatures that can replace those of I_2 (covariant input, contravariant output). Symmetrically to the provided interfaces, the required interfaces may have a canonical description. The difference consists in encoding the fact that a component that does not have a required interface can replace a component that has this interface.

Figure 15 shows the lattice of required map interfaces. It reveals that:

- `IGPSMap` and `ILocMap` can replace a (new) required interface containing the three signatures represented in concept C_2: `findMap(Location, Location):Map`, `extractRoadNetwork(Map):Hypergraph`, and `map(Location):ZoomableMap`

[7]All lattices are available at: http://code.google.com/p/dicosoft/downloads/list.

- all required map interfaces can replace `ILocZoomMap`.

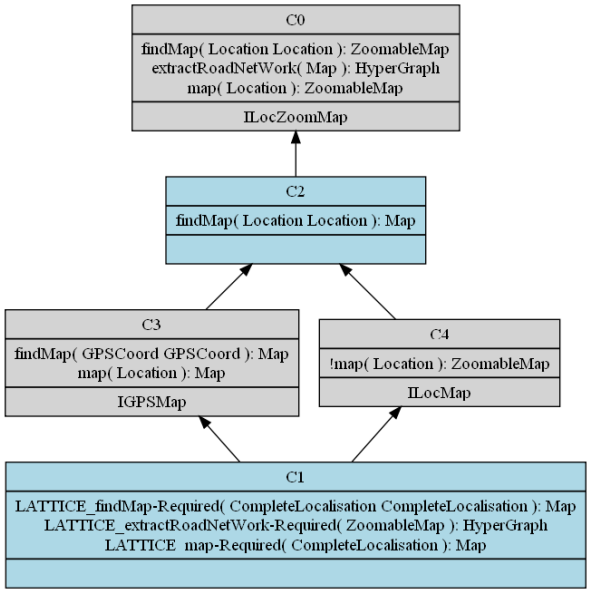

Figure 15 – The lattice of required interfaces about maps

4.3.3 Classifying Components

Finally, we classify components. A component Cp_1 can replace a component Cp_2 if it offers more and requires less from its environment:

- Cp_1 may have more provided interfaces,
- Each provided interface of Cp_2 can be replaced by one of the provided interfaces of Cp_1.
- Cp_1 may have fewer required interfaces,
- Each required interface of Cp_1 can replace one of the required interfaces of Cp_2.

The result of component classification for our example (components of Figures 1 to 7) is the lattice shown in Figure 16. Component concepts also may have a canonical description. The canonical description of a component is composed of the union of the most specific provided interfaces and of the most specific required interfaces from the intent of the concept ('most specific' refers here to their position in their respective concept lattices). Figure 6 shows a specialization order extracted from the concept lattice of Figure 16.

4.4 Discussion

We have presented a basic approach for organizing components in a substitutability-based structure. There are many possible variations on the connection and substitution model, depending on the component model and the adaptation effort that can be admitted at substitution and connection time. A more strict connection model could

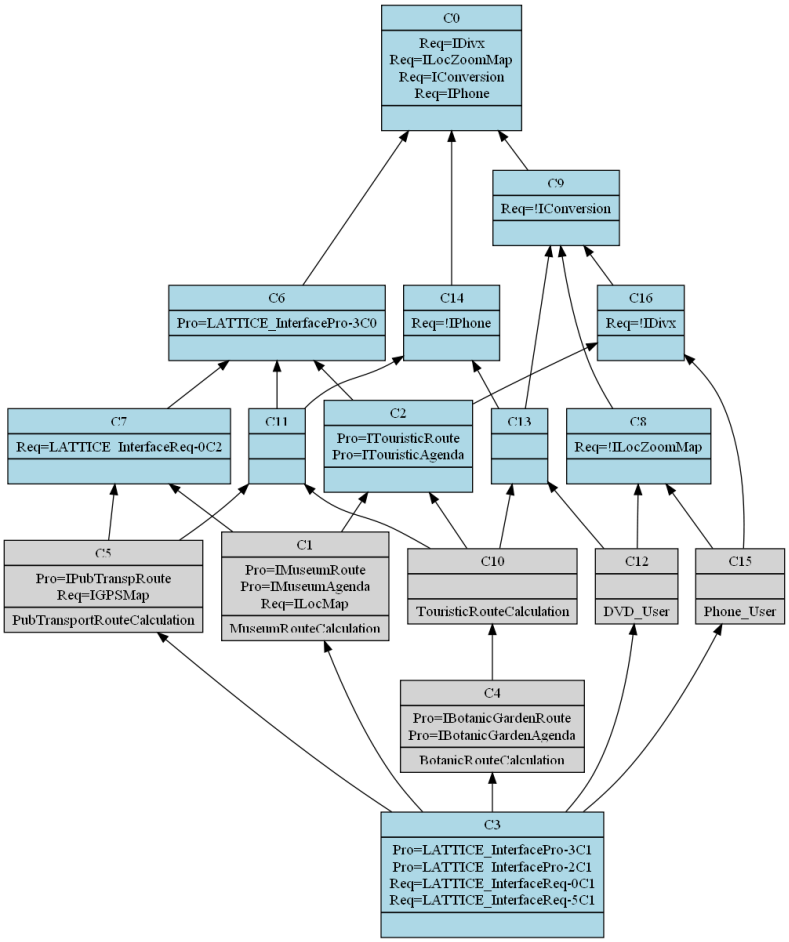

Figure 16 – Component lattice

constrain the order and the number of the parameters, or the invariance in functionality signatures (same parameter types in the same order and same return type). Fortunately, Formal Concept Analysis offers many different encoding schemes to deal with most of the situations [GW99]. All these variations can be understood as a parameterization of the generic classification process. Given that we rely on functionality names and type hierarchies of the parameter types, our approach best suits components that come from the same component provider.

One of the critical issues concerns the use of negation in the encoding. We use interfaces for discussion, but this also applies at the levels of parameters and functionality signatures. To respect the substitution principle and avoid substitution of a component by another component which has more required interfaces, we have seen that it is sometimes necessary to encode that a component does not have an interface. This is often the case when two components share some interfaces in the same interface lattices and one ($Comp_1$) has required interfaces (*e.g.*, `Req=`I) that the other ($Comp_2$) has not. A minimum needed to respect the substitution principle is encoding (inferring) the fact that $Comp_2$ does not have (at least) one of the interfaces

in the lattice that contains I.

An example is the top interface that we used in the illustrative example. For example, `DVD_User` and `Phone_User` components do not have interfaces to deal with `Map`, and this can be encoded as a negation by associating the `Req=!ILocZoomMap` characteristic, where `ILocZoomMap` is the top of the lattice of required interfaces about `Map` (Figure 15). To be more precise, `DVD_User` and `Phone_User` do not require `ILocZoomMap` (characteristic `Req=!ILocZoomMap`). This means that they can replace a component that requires `ILocZoomMap` (if no other characteristics contradict this replacement). To encode this in a formal context, this means that a component that owns `Req=!ILocZoomMap` should also own `Req= ILocZoomMap` (we call this an inference), to allow the replacement mentioned above. We can observe in Figure 16 that using this encoding, `DVD_User` in concept `C12` extent and `Phone_User` in concept `C15` extent have `Req=!ILocZoomMap` and by inheritance they own `Req= ILocZoomMap` (from concept `C0`) because of this encoding. `DVD_User` and `Phone_User` would be able to replace a component in the extent of concept `C0` (if such component existed in our example, which is not the case).

To increase the number of substitution opportunities, it is better to encode the fact that $Comp_2$ does not have any of the interfaces in the lattice that contains I and infer the positive counterpart. In this case, in our example, `DVD_User` and `Phone_User` would have characteristics corresponding to negation of each interface from the lattice of Figure 15: `Req=!ILocZoomMap`, `Req=!LATTICE_InterfaceReq-0C2` `Req=!IGPSMap`, `Req=!ILocMap`, (we can omit the bottom). Then to apply the same principle, extended to all interfaces, `DVD_User` and `Phone_User` are assigned the positive counter-parts: `Req=ILocZoomMap`, `Req=LATTICE InterfaceReq-0C2` `Req=IGPSMap`, `Req=ILocMap`. Now `DVD_User` and `Phone_User` can replace a component that requires `ILocZoomMap` (as in minimal negation encoding), but also a component that requires `IGPSMap`, offering more substitution possibilities (as previously, if no other characteristics contradict this replacement).

Thus, coming back to the general case, such encoding allows the component to substitute (if the remainder of its description is adequate) for a component that has one (any) interface in the lattice that contains I. An unwanted consequence of encoding the negation is that the lattice size rapidly grows when we encode all the possibilities for substitution, because many concepts appear that represent the sharing of not owned characteristics (parameters, signatures, interfaces). Thus, introducing negation implies that we have to know all possible combinations among properties, and this makes it difficult to manage in a dynamic context, where components could appear and disappear in a directory. In the illustrative example, we chose to encode the minimum needed and this issue does not change critically the resulting lattices. In particular, this does not change the classification of the initial components. To sum up, in order to avoid erroneous substitutions, it is necessary to encode at least that when a component does not have an interface, it does not own the top lattice interface. To provide more substitution opportunities, it is useful to infer more interfaces from the lattice.

5 Validation

In this section, we describe the Dicosoft prototype tool which has been developed in order to implement the classification method (Sect. 5.1) and to make experiments on real-world component libraries. We present then an experimentation of the classification

method where we wanted to answer the following research question: "Is the classification method scalable over real-world component libraries of different sizes?" We have quickly found out, by testing DICOSOFT, that the method needs some optimizations to be able to classify large component libraries. We proposed two variations to the classification method (that we will consider, in the remaining text, as classification methods *per se*). In addition to optimizing the classification process, these methods provide complementary organization views on components (Sect. 5.2). In the last part of this section, we present the experimental evaluation which has been conducted on three Fractal component libraries of different sizes (Sect. 5.3).

5.1 The DICOSOFT tool

DICOSOFT is a Java prototype tool that we developed in order to implement the component classification method. It analyzes repositories of Fractal components described in Fractal ADL[8] and implemented in Java with Julia (the reference implementation of Fractal)[9]. Figure 17 shows the most important modules in DICOSOFT: the `LibraryParser` and the `LibraryClassifier`.

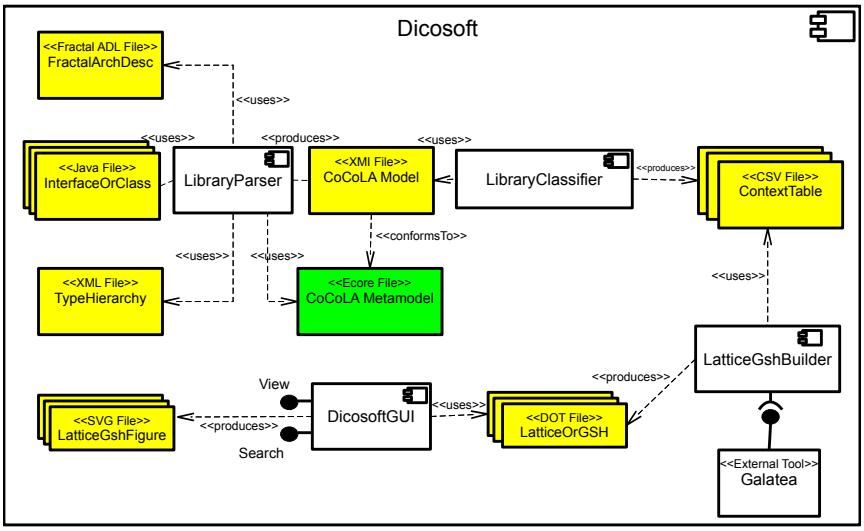

Figure 17 – A Simplified Architecture of DICOSOFT

The `LibraryParser` takes as input all the following files:

- Fractal ADL (XML-based) files describing components, by specifying the interfaces they provide and require and the (nested) descriptors of their potential internal (encapsulated) components.
- Java source and class files containing interfaces and classes implementing the components. If a flat structure of files is not available, JAR files can also be parsed by DICOSOFT to extract interface and class files.

[8] Fractal ADL in OW2 website: http://fractal.ow2.org/tutorials/adl/
[9] Julia in OW2 website: http://fractal.ow2.org/julia/

- An XML file containing the type hierarchy (here, types are interfaces, classes and component types).

This tool provides as output an intermediate XML file containing a merge of all the parsed information. This enables to have a single file with all the information necessary for the classification. Besides, this allows us to decouple Dicosoft from the Fractal ADL syntax. Concretely, this XML file is an XMI (XML Metadata Interchange) document that complies to the CoCoLa (Components in Concept Lattice Analysis) metamodel [AAF+09].

The CoCoLa metamodel generalizes the Fractal component model by adding information about component implementations. It summarizes all data on component architecture descriptions. This metamodel has been implemented as an Ecore model on Eclipse [Ecl09]. Thanks to the EMF (Eclipse Modeling Framework) plugin [Ecl09], we obtained an API to parse and generate XMI files that comply to the CoCoLa metamodel.

The `LibraryClassifier` takes as input the XMI file generated by the first module and generates context tables, according to the classification variant selected by the user. It then uses an external tool developed by our team, named eRCA[10], to generate lattices or AOC-POSETs.

DICOSOFT includes a graphical user interface which enables end-users to choose a component library. It then generates classifications according to the selected classification method (variant). It provides views of the classifications (or exports them as images) and also makes it possible for the user to specify queries to analyze classifications. These queries are names of components, interfaces or functions. DICOSOFT returns a list of possible substitutes for the chosen architectural elements. Figure 18 shows a screenshot of our prototype tool. For more details about DICOSOFT, the reader is invited to visit: `http://code.google.com/p/dicosoft/`.

5.2 Implementation of the Approach: Original Method and Optimizations

To show the applicability of our approach, we have developed three different implementations named as `complete`, `fast` and `smart`. The last two ones were implemented because of different problems generated by the first one to validate our approach.

Complete. Early experiments were developed with the initial classification method (called `complete`). However, the serious problems arrived when building the lattices due to time and memory space reasons. These problems showed us that we need to find techniques to make the approach scalable. One answer we found was with the computation of the AOC-POSET[BGH+14] rather than building the lattices. In particular, in the AOC-POSET, most of the useless concepts that group only negations of characteristics have disappeared. Following, we also explain two other classification methods that we experimented on real-world libraries to check their feasibility and evaluate their efficiency. Furthermore, they provide an additional usage scenario to component substitute search, which is component refactoring that is detailed later.

Required/Provided variant (fast). Our first variant method separates the required and provided parts of the components. Lattices of functionality signatures and of interfaces are built as in the initial approach. Component description is separated into two parts, generating two lattices, for the required and provided parts respectively.

[10] http://code.google.com/p/erca/

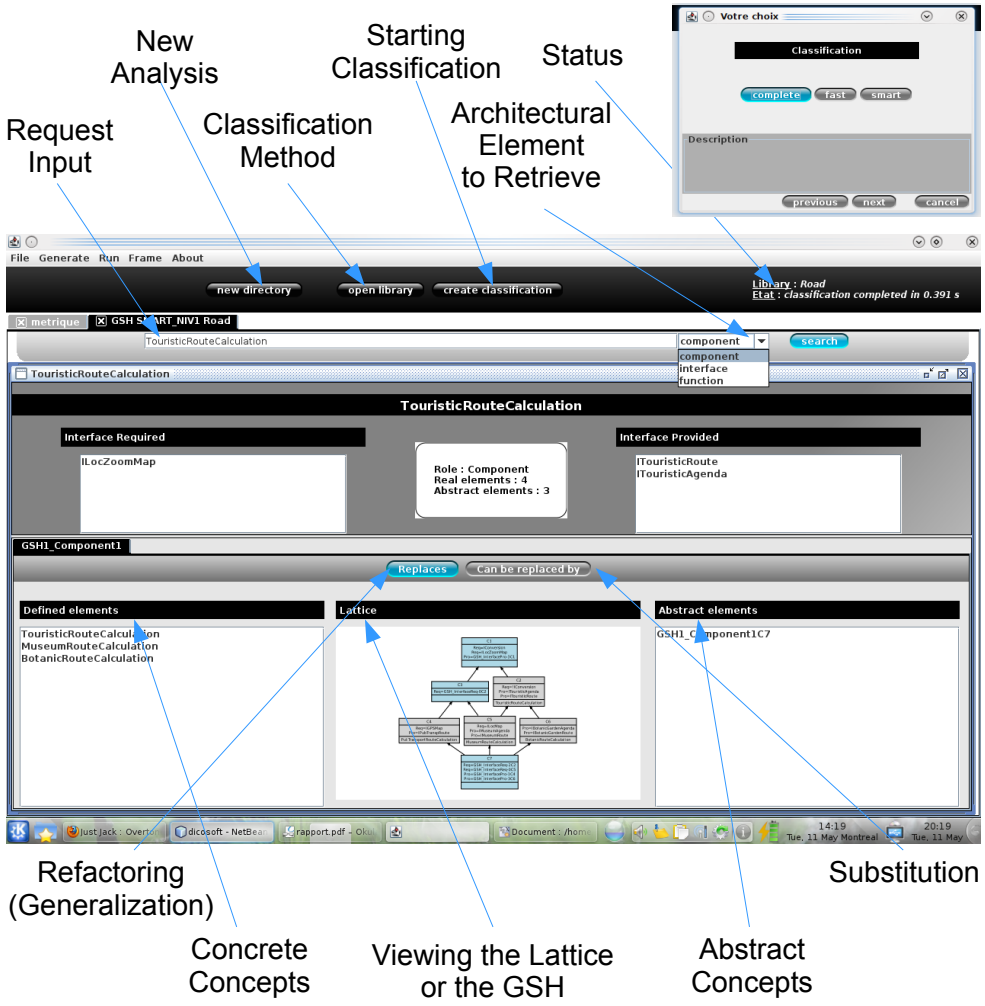

Figure 18 – Screenshot of Dicosoft's GUI

The provided-part context contains the components associated with the provided interfaces they own. The required-part context contains the components associated with the required interfaces they own, and it is also filled up by inferences as in the complete approach.

This `fast` component classification approach results in smaller structures on which it is easier to reason, identifying at first in the provided part what a designer needs as a primary functionality, and then, in a second step, checking if the environment is able to provide what the chosen components require. The counterpart of this separation is that, now, determining if a component can be a substitute for another component, requires the analysis of the relations between the two components in the two lattices. To ease this, new lattices are generated that merge this information. Each merged lattice contains the components that can replace a given component in the two, required and provided, lattices where this component appears.

Bipartite variant (smart). The motivation of this second variant is the observation that subgroups of components that share a set of functionalities can often be isolated when analyzing the contents of component repositories. If there is not (or little) intersection between these subgroups, it might be preferable to build a separate lattice for each subgroup. Indeed, it is uninteresting to classify together two components that will not (or likely not) be a substitute for the other.

To find these subgroups, we build a bipartite graph associating components to their functionality names associated with a direction. Provided and required directions are noted as either `_Pro` or `_Req` suffixes to functionality names. Figure 19 depicts this bipartite graph for our example.

A simple way of cutting the bipartite graph into smaller graphs consists in using its connected subgraphs. It is unfortunately inefficient in the example, because there is only one connected subgraph in this graph. This method was not interesting either when applied on the analyzed repositories we experimented on, because we found very large connected subgraphs.

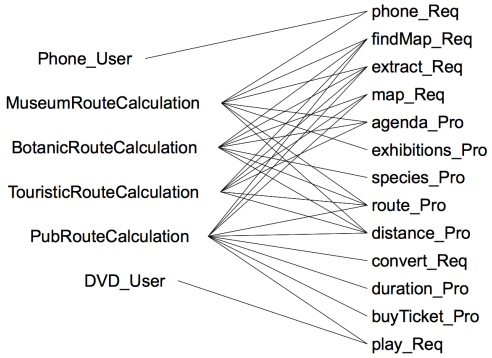

Figure 19 – Bipartite graph connecting components and their directed functionality names

In the solution we chose, the bipartite graph is decomposed into particular bipartite subgraphs such that two components belong to the same subgraph if they share at least one directed (required or provided) functionality name. The component context is then divided into smaller contexts, each of which corresponds to a subgraph. Some components can appear in several subgraphs. In our example, this gives three small contexts. Substitution can be done as in the complete approach, by looking for the respective positions of the two components in the lattice that contains both of them.

5.3 Experimentation on three Fractal component libraries

We experimented the approach presented in our paper on three different Fractal component libraries of various sizes: Swing (smallest in terms of number of components), Rmi and Dream (largest). Figure 20 depicts the size of each analyzed library.

In the next subsections, we highlight different operations offered by our classification approach such as component search for substitution or component refactoring, and we evaluate scalability of the classification approach using these three libraries. In the first subsection, we show how a software architect can retrieve a possible substitute for a failed component in a library classification. This library is Fractal Swing [11]. We provide some metrics on the library and on the Galois Sub-hierarchies (AOC-poset)

[11] Fractal Swing on the OW2 Website: http://fractal.ow2.org/java.html

	Swing	Rmi	Dream
#components	41	57	221
#provided interfaces	19	21	70
#required interfaces	4	20	59
#provided functions	286	19	70
#required functions	82	18	56

Figure 20 – Metrics on the Fractal libraries Swing, Rmi and Dream

obtained with the three different classification variants. We discuss which is the variant that is best suited for this task (component substitute search in a relatively small library). In the second subsection, we expose the classification of the Fractal Rmi library[12]. We illustrate a refactoring use case on this classification and explain how a component designer can extract some interesting component descriptions which are more abstract than the ones that already exist in the library. In this case, we use also a relatively small-sized library of 57 components (41 components for the first library). In the last subsection, we experiment the approach on a library of a larger size, Dream[13]. We show how the last classification variant is the most useful to process large libraries of hundreds/thousands components.

Use Case #1: "Small library & Substitute search". Fractal Swing is a library providing a set of Fractal components to build graphical user interfaces.

Table 1 shows the results obtained using the different classification methods (first three columns whose common header is `Swing`). Even if this library contains only 41 components, the classification computation times are high comparatively to other libraries. The minimal total computation for classifying this library is 104 seconds with the Fast method (to be compared with the minimal time for classifying the second library, which is 357 milliseconds). This is mainly due to the structure of this library. In Fractal Swing, we observed that there is a high number of functions per interface (*e.g.,* for the provided part, 286 functions in only 19 interfaces). In addition, most of the types (interfaces and components) inherit from each other. This structure increases computation time of relations between objects and attributes needed to generate the AOC-poset. However, we chose to present this library because it makes substitution easier to understand.

The use case concerns a software architect whose role is to assemble components in order to build an application. During maintenance, we suppose that a component in the application fails (a bug might be detected while running the application, for example). Dicosoft can thus help the architect to find a new component in the library which can play the same role as the failed one. We take a concrete example of the architect of Fractal Gui[14], who used Fractal Swing components to build her/his application. Fractal Gui helps software architects to graphically design Fractal software architectures. Let us suppose that the `JPanelImpl` component fails (it does not perform correct graphical rendering). Through the classification of the Fractal Swing library using the `fast` method, we see which components can replace it in Fractal Gui. We chose to use the `fast` method because it is the best for component substitution. We can observe in Table 1 that the average computation time for finding a substitute is

[12]Fractal RMI on the OW2 Website: http://fractal.ow2.org/fractalrmi/index.html
[13]Dream library on the OW2 Website: http://dream.ow2.org/
[14]Fractal Gui on OW2 Website: http://fractal.ow2.org/fractalgui/index.html

	Swing			Rmi			Dream		
	Complete	Fast	Smart	Complete	Fast	Smart	Complete	Fast	Smart
Number of AOC-poset	1	2	1	1	2	12	1	2	31
Max. number of concrete concepts in AOC-poset	41	49	29	57	57	8	221	221	53
Min. number of concrete concepts in AOC-poset	41	49	29	57	57	1	221	221	1
Av. number of concrete concepts in AOC-poset	41	49	29	57	57	4	221	221	7
Max. number of abstract concepts in AOC-poset	12	0	11	20	0	8	75	0	24
Min. number of abstract concepts in AOC-poset	12	0	11	20	0	0	75	0	0
Av. number of abstract concepts in AOC-poset	12	0	11	20	0	2	75	0	4
Total number of generated concrete concepts	41	82	29	57	114	53	221	442	141
Total number of generated abstract concepts	12	0	11	20	0	29	75	0	225
Total number of generated concepts	53	82	40	77	114	82	296	442	366
Av. number of substitutions per component	10	7	9	16	4	5	72	4	13
Max. computation time for finding a substitution	874 ms	50 ms	797 ms	5 ms	74 ms	2 ms	116 ms	577 ms	8 ms
Min. computation time for finding a substitution	34 ms	3 ms	40 ms	1 ms	1 ms	1 ms	1 ms	2 ms	1 ms
Av. computation time for finding a substitution	198 ms	9.89 ms	221 ms	1 ms	7 ms	1 ms	3 ms	67 ms	1 ms
Computation time for classifying functions	5.5 s	5.5 s	5 s	363 ms	212 ms	106 ms	1.5 s	916 ms	815 ms
Computation time for classifying interfaces	18.5 s	18.5 s	18.5 s	29 ms	16 ms	16 ms	292 ms	62 ms	61 ms
Computation time for classifying components	166 s	80 s	158 s	1 s	255 ms	235 ms	51 s	986 ms	2023 ms
Total computation time for classification	189 s	104 s	182 s	1484 ms	500 ms	357 ms	53 s	1964 ms	2899 ms

Table 1 – Metrics on the classifications of the Fractal libraries Swing, Rmi and Dream

almost 10 milliseconds, while for the other two methods it is around 200 milliseconds. The reason is that the `Fast` method generates a minimal number of intermediate abstract concepts in the AOC-poset. The AOC-poset contains only components which are substitutable with each other.

The obtained AOC-poset[15] contains both required and provided parts, so it has been produced by merging the required and the provided AOC-poset built by the `fast` method. Among 41 components available in the Fractal Swing library, the AOC-poset proposes only 7 components as possible candidates for substitution. This reduces of almost 82% the size of the search space. Among 7 components that are proposed to the architect, she/he will choose the one which is the most relevant to her/his needs, like the component `JScrollPaneImpl` which can replace `JPanelImpl` (*cf.* Figure 21).

For this use case (small library & substitution), the user can choose either the `Complete` or the `Fast` method. It is preferable to use the `Fast` method first, because, as we have seen previously, even if the library is small its structure can make the

[15] http://code.google.com/p/dicosoft/downloads/detail?name=org2.png&can=2&q=

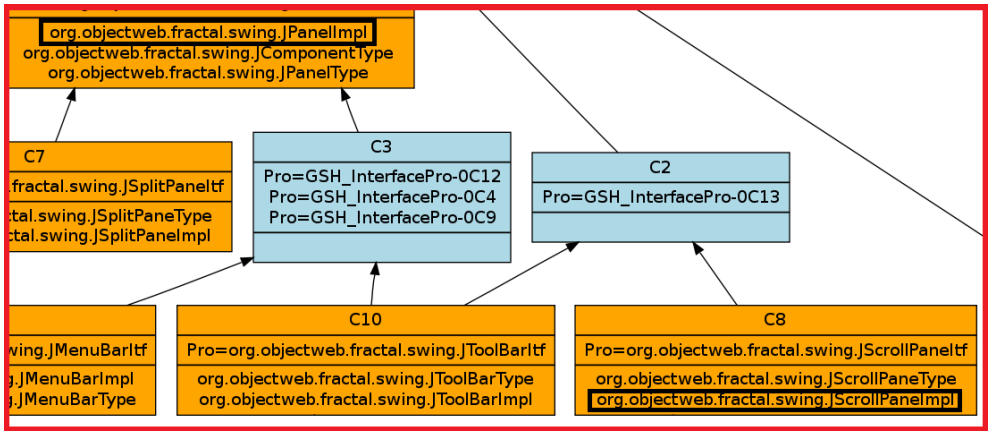

Figure 21 – Zoom on the AOC-poset for the substitution of the `JPanelImpl` component

computation time for building the classification high. The user can however, by using this method, miss some substitutes, because the `Fast` method provides generally less substitutes than the `Complete` method: for the Swing library, 7 substitutes in average with the `Fast` method, compared to 10 substitutes in average with the `Complete` method (with the other two libraries the difference is more important: 4 vs. 16 and 4 vs. 72, see Table 1). In the case where the suggested substitutes do not satisfy the user, she/he can use the Complete method to retrieve more substitutes.

Use Case #2: "Small library & Refactoring". Fractal Rmi is a library of Fractal components which helps in building distributed component-based applications. Figure 20 shows some measures made on this library, while Table 1 shows the results obtained when applying the three classification methods using Dicosoft. Dicosoft helps a component designer in discovering new component definitions which generalize (which make more abstract) the definitions of components that already exist in the library. These new components are more substitutable because there are many components that are more specific to them.

We use the `complete` method to make this refactoring that aims to extract more abstract component definitions. The `complete` method classifies all components in a single AOC-poset to search for a more abstract definition for the two components presented in Figure 22.

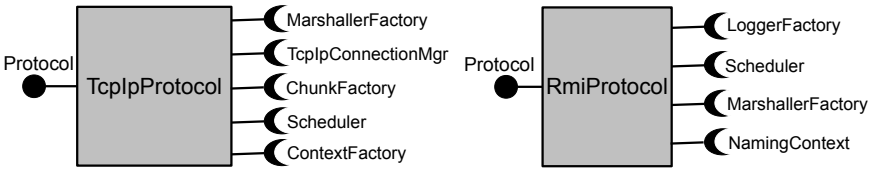

Figure 22 – Two concrete components from the Fractal RMI library

Figure 23 depicts a zoom on the AOC-poset of the Fractal Rmi library. We see that concept 19 is a new abstract definition which factorizes the definitions of `TcpIpProtocol` and `RmiProtocol`. When implementing this component, we obtain a

new hierarchy which is illustrated in Figure 24.

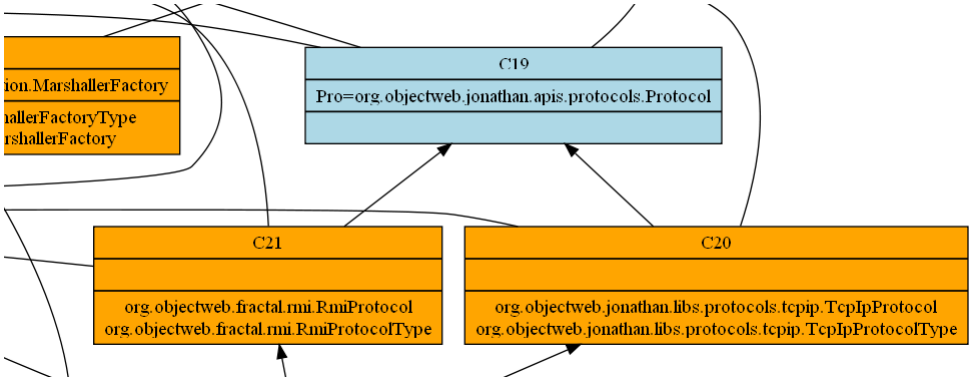

Figure 23 – Zoom on theAOC-poset of the Fractal Rmi library

In the Fractal Rmi library, there is no existing component which can be replaced by components `TcpIpProtocol` or `RmiProtocol`. The implementation and integration of this newly suggested component in the library thus adds two possible substitutions (`TcpIpProtocol` or `RmiProtocol` can replace the new component). Thereby, such component addition increases the overall flexibility of components from the library. In this use case, we took the example of an abstract component which generalizes two components, but if we further analyzed the generated AOC-poset, we would find other generalizations.

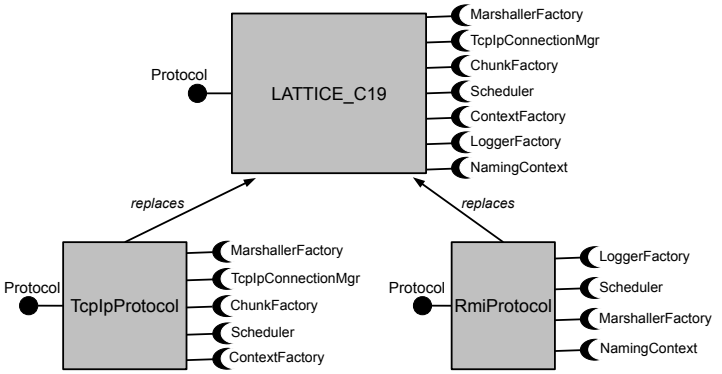

Figure 24 – Hierarchy after adding the new component

With the `complete` method, all components are classified in a unique AOC-poset. This classification has the drawback that if we use large libraries (of more than 100 components), the obtained classification becomes very large. If we take the classification of Fractal Rmi, we see that for 57 components, we obtained 77 concepts (57 concrete / 20 abstract) in a single classification. With the `fast` method, we obtained 114 concrete concepts, and no abstract concepts, for the same library. For Fractal Swing (which contains 41 components), we obtained 53 concepts (41 concrete / 12 abstract) with the `complete` method and 82 concepts with the `fast` method. To compare the classification size, one has to consider that, for "Fast" method, two AOC-posets are computed, one for the provided part, the other for the required part (each

one contains 57 concepts). For "Smart" method, there are also several AOC-posets that are small (they have at most 16 concepts).

For this particular case of refactoring, and with libraries of a relatively small size (less than 100 components), the designer should use the `Smart` method, instead of the two others. The total time for the computation of classifications is less than the `complete` method. In addition, the number of abstract concepts is generally greater (it is almost the same for the previous library, Fractal Swing: 11 abstract concepts with the `smart` method and 12 with the `complete` one, but it is greater for the other libraries: 29 vs. 20 and 225 vs. 75).

Use case # 3: "Large library classification". The third study has been conducted on Dream[16], which is a library of Fractal components designed to build communication middleware. It provides a set of components and tools to specify, configure and deploy a middleware implementing many communication paradigms. Figure 20 depicts some metrics on this library. Dream is the largest Fractal component libray that we have found till now. It contains 221 components.

When classifying larger libraries such as the Dream library, we observe that components belong to distinct domains, such as communication protocols or management. Building a single AOC-POSET for such libraries is useless for multiple reasons: i) complexity of the result (296 concepts for 221 Dream components), ii) higher computation time (53 seconds for the Dream library), and iii) less chance to find a substitute for a given component belonging to a given domain in the other domains, which are in the same classification. Using the `smart` method, we identify the different domains in a library and classify these domains separately. In the case of Dream, we identified 31 domains. The main domain is relative to protocols, coding and serialization of messages. It contains 56 components. The total computation time of all the classifications is approximately 3 seconds, compared to 53 seconds with the `Complete` method. For relatively large libraries, the user should prefer the `Smart` method for refactoring and the `Fast` method for substitution.

To conclude this (validation) section, it is important to state the threats to the validity of our experimentation. First, the internal validity is related to the confidence that we have in the correctness of the component libraries we have used. It is worth mentioning that each of these libraries has been designed by the same development team. This can be a bias in the experimentation, since components that can substitute each other can be put close the ones from the others in the type hierarchy, as leaves of the same node. This makes them appear necessarily as substitutes in the classification. Thereby they can be easily and quickly retrieved. However, this aspect (that the library has been designed by the same team) is an important element in the construction of accurate classifications, because signatures of functionalities provided/required by components are likely to share the same types or to be defined by using type variance (since types are defined by the same team). If components are defined separately, by different teams, the types they use in their signatures are likely to be different. This requires type matching to identify relationships between types, which do not provide always accurate results.

Second, the external validity concerns the capacity of our experimentation to be generalized to other component libraries. Even if the experimentation has been conducted on Fractal libraries, our classification method can be easily used with any *object-oriented component model* (as categorized in [CSAC11]). The only component

[16] http://dream.ow2.org/

that should be replaced in DICOSOFT is the `LibraryParser`.

6 Conclusions

This paper presented a novel contribution to automate software component classification composed of both a methodology and its prototype implementation. Its objectives are to ease queries in component directories and to improve reusability and substitution. Through the analysis of component external descriptions (functionality signatures, provided and required interfaces), we proposed a three step classification process that uses Formal Concept Analysis. Summarizing, the steps are as follows: In the first step, data type hierarchies are used to calculate functionality signature lattices that encode the substitutability relationship. Then, functionality signature lattices are used to calculate provided or required interface lattices. To finish, interface lattices are used to calculate component lattices that the architects can query to search for a component to connect or to substitute for another.

We also present three strategies to tame complexity: a *complete* strategy, and two divide-and-conquer strategies named *fast* and *smart*. We have built different prototypes to test the different strategies on real component libraries and discussed their adequateness and scalability.

Regarding future work, we propose several approaches. Firstly, variants on the substitutability relationships could be investigated. For example, we could take exception types (thrown by functionalities) into account when analyzing functionality signatures. Automating the generation of simple adapters starting from lattices would also be interesting for architects. We would like to extend the classification criteria to be able to compare non homonymous functions using natural language-based similarity measures. This would make it possible to compare components that do not share the same vocabularies (for example, components that come from different providers). The possibility of filtering results with non-functional attributes or information gathered on component usage (ranking or opinion mining, capitalizing previously made adaptations, structuring architect folksonomies, etc.) would also greatly increase user confidence and directory structure. Another subject to explore is to analyze how we can adapt the proposed methodology to other component-based systems that have a description like in ADLs languages. As a complement to the presented work, we can also analyze if the methodology impact in QoS or behavioral aspects of components.

References

[AAF+09] Nour Alhouda Aboud, Gabriela Arévalo, Jean-Rémy Falleri, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Automated architectural component classification using concept lattices. In *Proc. of WICSA/ECSA'09*, pages 21–30, Cambridge, UK, September 2009. IEEE CSP. doi:10.1109/WICSA.2009.5290788.

[ABC+06] Lerina Aversano, Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, and Damiano Distante. Using concept lattices to support service selection. *International Journal of Web Services Research (IJWSR)*, 3:32–51, 2006.

[ADH+11] Zeina Azmeh, Maha Driss, Fady Hamoui, Marianne Huchard, Naouel Moha, and Chouki Tibermacine. Selection of composable web services

driven by user requirements. In *Proc. of The 9th IEEE International Conference on Web Services (ICWS'11), Applications and Experiences Track*, pages 395–402, Washington DC, USA, July 2011. IEEE Computer Society Press. doi:10.1109/ICWS.2011.47.

[AHM+11] Zeina Azmeh, Fady Hamoui, Nizar Messai, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Backing composite web services using formal concept analysis. In *Proc. of the 9th International Conference on Formal Concept Analysis (ICFCA'11)*, volume 6628 of *Lecture Notes in Computer Science*, pages 26–41, Nicosia, Cyprus, May 2011. Springer-Verlag. doi:10.1007/978-3-642-20514-9_4.

[AHT+08] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Wspab: A tool for automatic classification and selection of web services using formal concept analysis. In *Proc. of the 6th IEEE European Conference on Web Services (ECOWS'08)*, pages 31–40, Dublin, Ireland, November 2008. IEEE Computer Society Press. doi:10.1109/ECOWS.2008.27.

[All06] OSGi Alliance, editor. *OSGi Service Platform, Core Specification, Release 4, Version 4.2*. aQute Publishing, 2006.

[BGH+14] Anne Berry, Alain Gutierrez, Marianne Huchard, Amedeo Napoli, and Alain Sigayret. Hermes: a simple and efficient algorithm for building the AOC-poset of a binary relation. *Annals of Mathematics and Artificial Intelligence*, may 2014. doi:10.1007/s10472-014-9418-6.

[BJAR11] Nabil Belaid, Stéphane Jean, Yamine Aït Ameur, and Jean-François Rainaud. An ontology and indexation based management of services and workflows application to geological modeling. *IJEBM*, 9(4):296–309, 2011. URL: http://ijebm.ie.nthu.edu.tw/IJEBM_Web/IJEBM_static/Paper-V9_N4/A02.pdf.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, LNCS 173, pages 51–67. Springer, 1984. doi:10.1016/0890-5401(88)90007-7.

[Cas95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995. doi:10.1145/203095.203096.

[CLL+10] Stéphanie Chollet, Vincent Lestideau, Philippe Lalanda, Diana Moreno-Garcia, and Pierre Colomb. Heterogeneous service selection based on formal concept analysis. In *6th World Congress on Services*, pages 367–374, July 2010.

[CSAC11] Ivica Crnkovic, Severine Sentilles, Vulgarakis Aneta, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, Sep 2011. URL: http://dx.doi.org/10.1109/TSE.2010.83, doi:10.1109/TSE.2010.83.

[DHT+08] Nicolas Desnos, Marianne Huchard, Guy Tremblay, Christelle Urtado, and Sylvain Vauttier. Search-based many-to-one component substitution. *Journ. of Software Maintenance and Evolution: Research and Practice*, 20(5):321–344, September/October 2008. doi:10.1002/smr.377.

[DMJ+10] Maha Driss, Naouel Moha, Yassine Jamoussi, Jean-Marc Jézéquel, and Henda Hajjami Ben Ghézala. A requirement-centric approach to web service modeling, discovery, and selection. In *Proc. of the 8th International Conference on Service-Oriented Computing (ICSOC 2010)*, pages 258–272, 2010. doi:10.1007/978-3-642-17358-5_18.

[Ecl09] Eclipse. Eclipse modeling framework (emf). Eclipse Board Web Site : http://www.eclipse.org/modeling/emf/, 2009.

[Edw00] W. Keith Edwards. *Core JINI*. Sun Microsystems Press Java. Prentice Halll, second edition, 2000.

[EH07] Clement Escoffier and Richard S. Hall. Dynamically adaptable applications with iPOJO service components. *Software Composition*, pages 113–128, 2007. doi:10.1007/978-3-540-77351-1_9.

[Fis98] Bernd Fischer. Specification-based browsing of software component libraries. In *Proc. of the 13th IEEE int. conf. on Automated Software Engineering (ASE'98)*, pages 74–83, 1998. doi:10.1023/A:1008766409590.

[GFS08] Bart George, Régis Fleurquin, and Salah Sadou. A component selection framework for cots libraries. In *Proc. of the Symposium on Component-Based Software Engineering (CBSE'08)*, pages 286 – 301. LNCS 5282, October 2008. doi:10.1007/978-3-540-87891-9_19.

[GK13] Suresh Chand Gupta and Ashok Kumar. Reusable Software Component Retrieval System. *International Journal of Application or Innovation in Engineering and Management*, 2(1), 2013.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[HA06] Oliver Hummel and Colin Atkinson. Using the web as a reuse repository. In Maurizio Morisio, editor, *Reuse of Off-the-Shelf Components, 9th International Conference on Software Reuse, ICSR 2006, Turin, Italy, June 12-15, 2006, Proceedings*, volume 4039 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2006.

[HHNV13] Mohamed Rouane Hacène, Marianne Huchard, Amedeo Napoli, and Petko Valtchev. Relational concept analysis: mining concept lattices from multi-relational data. *Ann. Math. Artif. Intell.*, 67(1):81–108, 2013.

[HJA08] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008. doi:10.1109/MS.2008.110.

[HTL+08] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. SLCA, composite services for ubiquitous computing. In *Proc. of the International Conference on Mobile Technology, Applications, and Systems*, pages 1–8, New York, New York, USA, 2008. ACM Press. doi:10.1145/1506270.1506284.

[Hum08] Oliver Hummel. *Semantic Component Retrieval in Software Engineering*. PhD thesis, Fakultat fur Mathematik und Informatik, Universitat Mannheim, 1 2008.

[ISO98] ISO/IEC. ODP Trading Function Specification ISO/IEC 13235-1:1998(E), December 1998. URL: http://webstore.iec.ch/.

[ITV04] Luis Iribarne, José M. Troya, and Antonio Vallecillo. A trading service for COTS components. *The Computer Journal*, 47(3):342–357, 2004. doi:10.1093/comjnl/47.3.342.

[Lin95] Christian Lindig. Concept-based component retrieval. In J. Köhler et al., editors, *IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, 1995.

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23:17–34, January 1987. doi:http://doi.acm.org/10.1145/62139.62141.

[LS00] Rosanna Lee and Scott Seligman. *JNDI API Tutorial and Reference: Building Directory-Enabled Java Applications*. Addison-Wesley Professional, 2000.

[LW94] Barbara Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994. doi:10.1145/197320.197383.

[MMGL01] Raphael Marvie, Philippe Merle, Jean-Marc Geib, and Sylvain Leblanc. Type-safe trading proxies using TORBA. In *Fifth Int. Symp. on Autonomous Decentralized Systems, ISADS, IEEE Computer Society*, pages 303–310, 2001. doi:10.1109/ISADS.2001.917433.

[MVA10] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley Professional, 1st edition, 2010.

[OMG00] OMG. Trading Object Service Specification (TOSS) v1.0. http://www.omg.org/cgi-bin/doc?formal/2000-06-27, 2000.

[OMG07] OMG. Unified modeling language superstructure, version 2.1.1 specification, document formal/07-02-03. Object Management Group Web Site: http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf, 2007.

[PHWZ05] Dunlu Peng, Sheng Huang, XiaoLing Wang, and Aoying Zhou. Management and retrieval of web services based on formal concept analysis. In *The Fifth International Conference on Computer and Information Technology (CIT'05)*, pages 269–275, Sept 2005.

[Pit01] Esmond Pitt. *Java.rmi: The remote method invocation guide*. Addison-Wesley Professional, July 2001.

[Sie00] Jon Siegel. *Corba 3: Fundamentals and Programming*. John Wiley & Sons Inc, 2nd revised edition, 2000.

[SR06] Benjamin Sigonneau and Olivier Ridoux. Indexation multiple et automatisée de composants logiciels. *Technique et Science Informatiques*, 25(1):9–42, 2006. doi:10.3166/tsi.25.9-42.

[SST10] Nedhal A. Al Saiyd, Intisar A. Al Said, and Ahmed H. Al Takrori. Semantic-Based Retrieving Model of Reuse Software Component . *IJCSNS International Journal of Computer Science and Network Security*, 10(7), 2010.

[Wil82] Rudolf Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, 83:445–470, September 1982. doi:10.1007/978-3-642-01815-2_23.

[YRZM10] Wei Yan, Francois Rousselot, and Cecilia Zanni-Merk. Component Retrieval Based on Ontology and Graph Patterns Matching. *Journal of Information and Computational Science*, 7(4), 2010. URL: `http://www.joics.com`.

[ZUV10] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric development and evolution processes for component-based software. In *Proc. of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*, pages 680–685. Knowledge Systems Institute Graduate School, 2010.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, 1997. `doi:10.1145/261640.261641`.

About the authors

Nour Aboud holds a PhD from Université de Pau et des Pays de l'Adour, France since 2012. During her PhD, she worked on a service-oriented integration of software component and organizational multi-agent models. She now is a research and development engineer at Items Media Concept, a private company of engineering and training in computer science near Bordeaux, France. Contact her at `nour.aboud@gmail.com`.

Gabriela Arévalo is full professor since 2011, and the main responsible for the Computer Science degree career since 2016 at Universidad Nacional de Quilmes (Buenos Aires, Argentina). She received her PhD in Computer Science in 2005. Her research interests are software reengineering mainly applied in object-oriented applications and using Formal Concept Analysis. She can be contacted at `garevalo@unq.edu.ar`.

Olivier Bendavid graduated from a computer science master from the University of Montpellier, France in 2010. During his master internship in Montreal (Canada), he worked on business intelligence in the cloud.

Jean-Rémy Falleri is currently associate professor at the Bordeaux INP and a member of the LaBRI laboratory where he is head of the software engineering research group. He received his Ph.D. degree in 2009 from the University of Montpellier 2 and his accreditation to supervise research in 2015. He has also worked in the RMoD research group of Inria Lille led by Stéphane Ducasse. His research interests lies in software engineering with a focus on software evolution and maintenance. Contact him at `falleri@labri.fr`, or visit `http://www.labri.fr/perso/falleri/`.

Nicolas Haderer graduated from a PhD in Computer Science, University of Lille in 2014. His PhD thesis focuses on Mobile Crowd Sensing, a new sensing paradigm based on the power of various smart devices to massively collect data. He now is technical lead of a decision support tool using crop models to predict growth and disease risks of wheat in ITK, a private company near Montpellier, France. Contact him at `haderer.nicolas@gmail.com`.

Marianne Huchard is full professor at University of Montpellier, France, since 2004. She received her PhD in Computer Science in 1992. Her research interests include theory and tools related to Formal Concept Analysis (FCA), and applications of FCA to Software Engineering, such as class hierarchy refactoring, interface extraction, or software product line reengineering. She can be contacted at `marianne.huchard@lirmm.fr`, or visit `http://www.lirmm.fr/~huchard/`.

Chouki Tibermacine is associate professor at University of Montpellier, France. He received his PhD in Computer Science in 2006 from the University of Southern Brittany, France and his accreditation to supervise research in 2018. His current research focuses on the promotion of reuse in software engineering practices through the development of new component models and languages, and the proposition of new methods for the migration of legacy software systems towards component- and service-based paradigms. Contact him at `Chouki.Tibermacine@lirmm.fr`, or visit `http://www.lirmm.fr/~tibermacin/`.

Christelle Urtado is associate professor at IMT Mines Ales, Alès, France. She received her computer science PhD in 1998 and her accreditation to supervise research in 2016. She has long been involved in research on component or service-based software engineering, focusing on component reuse and software composition and evolution. Component type issues are part of her research as they are a cornerstone of software evolution. Contact her at `Christelle.Urtado@mines-ales.fr`, or visit `http://www.lgi2p.mines-ales.fr/~urtado/`.

Sylvain Vauttier is associate professor at IMT Mines Ales, Alès, France. He received his computer science PhD in 1999 and his accreditation to supervise research in 2018. His work is focused on software architecture construction, evolution and reuse, encompassing multiple paradigms like components, services and agents and approches like IR, MDE and SBSE. Contact him at `Sylvain.Vauttier@mines-ales.fr`, or visit `https://sylvainvauttier.wp.imt.fr/`.

Acknowledgments Authors would like to warmly thank Nicolas Auboin and David Pallet who contributed to the development of the DICOSOFT tool.