Contributions to component-based software engineering – composition, reuse and evolution –

Christelle Urtado

I2S doctoral school, computer science speciality

University of Montpellier

Habilitation thesis

Contributions to component-based software engineering

– composition, reuse and evolution –

Christelle Urtado

IMT associate professor
LGI2P, école des mines d'Alès
Nîmes, France
Christelle.Urtado@mines-ales.fr
www.lgi2p.mines-ales.fr/~urtado/

Defended on November 8^{th}, 2016 in front of the jury composed of:

Mr. Antoine Beugnard	Professor, Telecom Bretagne	President
Mr. Jean-Michel Bruel	Professor, University of Toulouse	Rapporteur
Ms. Laurence Duchien	Professor, University of Lille	Rapporteur
Mr. Tom Mens	Professor, University of Mons, Belgium	Rapporteur
Ms. Marianne Huchard	Professor, University of Montpellier	Examiner
Mr. Mourad Oussalah	Professor, University of Nantes	Examiner
Mr. Salah Sadou	Associate prof., HDR, University of South Brittany	Examiner

A mon grand-père François,
A mes parents,
A Lucas,

Acknowledgements

C'est très chaleureusement que je remercie ici les personnes qui ont m'ont permis de présenter ce travail aujourd'hui.

Je remercie l'école des mines d'Alès, et surtout Yannick Vimont, son directeur de la recherche et directeur du LGI2P, pour m'avoir fourni le cadre de travail protégé dans lequel j'évolue depuis mes premiers pas de chercheur en 1994.

Je remercie les professeurs Jean-Michel Bruel, Laurence Duchien et Tom Mens pour avoir accepté de consacrer de leur temps à évaluer mon mémoire.

Je remercie les professeurs Antoine Beugnard, Marianne Huchard et Mourad Oussalah ainsi que le maître de conférences, HDR Salah Sadou pour avoir accepté de faire partie du jury de cette habilitation.

Je remercie tous mes collègues du site de Nîmes pour en avoir fait un environnement de travail agréable avec notamment Jacky, Vincent, Anne-Lise, Thomas, Nicolas, Françoise, Annie, François, Michel(s), Sylvie, Claude, Valérie.

Ce travail de recherche est, bien entendu, un résultat collectif. Je remercie tous les collaborateurs et doctorants qui y ont contribué.

Un merci tout particulier à Mourad qui m'a fait découvrir la recherche en 1994 et qui, après avoir supervisé mon doctorat, a toujours gardé un œil attentif sur la suite de ma carrière.

Un grand merci à Marianne, avec qui j'ai beaucoup de plaisir à travailler, pour ses conseils, sa présence et sa bienveillance sans faille.

Merci aussi à Sylvain pour ces années de remue-méninges partagés.

Merci à mes co-auteurs dont Christophe, Gabi, Guy, Chouki, Luc, Djamel et à mes doctorants et post-docs Frédéric, Nicolas, Yulin, Fady, Zeina, Matthieu, Guillaume, Rafat, Abderrahman dont la diversité des regards m'ont enrichie.

Merci également à toutes les personnes que j'ai croisées qui m'ont donné un coup de pouce ou qui ont contribué à me montrer le chemin.

Merci enfin à ma famille et à Lucas pour tout, et pour accepter avec humour que je "joue encore à l'étudiante" !

Contents

List of Figures

List of Tables

"We write to taste life twice, in the moment and in retrospection."

Anaïs Nin (1903 – 1977),
In favor of the sensitive man, and other essays.

Foreword

Since the beginning of my PhD in the end of year 1994, I have been working on various subjects related to modeling information systems and software using various abstractions: objects, components and services. This habilitation thesis is an opportunity for me to present these works altogether and try and provide the *big picture* I have in mind in which they fit.

During these 22 years, I indeed have worked on connected topics. The path followed is yet not linear. Even if my research is centered on the same few research questions I have been investigating on for this long, research themes were impacted by many factors: collaboration opportunities, research grant opportunities, projects, advised PhD students, etc. Subjects overlap. Others have long been idle and come back to the front. Altogether, they nonetheless form a whole I am happy to present today, a whole with blank zones I surely will be exploring in the future.

All these reasons led me to choose not to present my work chronologically. Not even in a sequence of subjects each studied with one of the PhD students I co-advised. Indeed, the works of some of them have spanned several subjects. Instead, I have chosen to organize this thesis thematically, mentioning where needed the contribution of whom I am talking about.

But before introducing my research subject to you, I would like to precise that the work I have been doing all these years would not have existed without the researchers and students with whom I collaborated. The work presented in this thesis is inherently collaborative. I owe my co-workers a lot. Even if the writing exercise will have me sometimes use "I", the reader should always understand "we".

I wish you a pleasant reading ...

Chapter 1

Introduction

"*The software industry is weakly founded, and (...) one aspect of this weakness is the absence of a software components subindustry*" [76]. This visionary declaration was made by M. Douglas McIlroy 45 years ago. A lot of progress have been made since then. A lot of theories, languages, techniques and tools have appeared, but, to some extent, this quote still applies. The forthcoming sections are going to define and introduce today's issues of software engineering before starting to focus on my field of research, that is component-based software engineering, and the three main activities of composition, reuse and evolution.

1.1 Software engineering issues

Software engineering is a young discipline the concern of which is to provide software architects and developers with languages, models, methodologies and tools that help build software in a controlled manner. Controlling the engineering process is a step towards maturity as enforced in other, more traditional, engineering domains such as electronics, mechanics or civil engineering, to cite a few.

As defined in the IEEE Standard Glossary of Software Engineering Terminology (IEEE610, 1990), "*software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*". The IEEE standard's vision is close to that of M. Douglas McIlroy: "*What I have just asked for is simply industrialism, with programming terms substituted for some of the more mechanically oriented terms appropriate to mass production*" [76].

The control of the software engineering process aims to rationnalize the way software is produced in order to reach (and be able to reproduce) better adequacy to user needs. This includes identifying and responding to user requirements but also being able to increase software complexity, reach better quality, longer lifetime (better maintainability), lower costs and shorter time to market.

As software pervades in our modern lives, it reaches more and more sectors of human activities and constraints are becoming harder.

- Software used to be critique in fields where human life could be threatened (in aeronautics for example). Today, as software pervades in cars, it is as much critique but also has to tackle the constraints of mass consumption goods (low costs, quicker development, frequent changes, etc.) that did not apply as sharply in more traditional critique applications.
- Software used to run in computer centers and be administered by highly qualified operating system and database experts. Today, as software pervades through intranets or in our mobile devices, it has to be more robust, less difficult to maintain, able to execute in various contexts (such as in open environments) and somehow should self-repair and self-adapt.
- Software used to support human activities (assist a plane pilot for example). Today metros are fully automatic and, in the near future, cars are going to be automated too. More burden relies on software in that it has to be perfect.
- Software used to be of intelligible size (as most projects did rely solely on new developments). Today, software better reuses previously developed parts and reaches higher complexity and size.

Software development is more critique and important as ever. Pressure on providing better methods to control its development is higher. Guaranteeing quality can no longer rely on the hero model (highly qualified experts who are very few to have the required capabilities). There must be a shift from crafts to engineering. Methods and tools are needed that makes high quality software development accessible to more software engineering professionals.

This can be achieved by sticking to up-to-date development methodologies or using the latest coding paradigms but, at the end, always amounts to:

(De)compose to manage complexity. It is a well known ancient principle that decomposing coarse problems into finer-grained ones to design their solutions and then compose back the solution to the coarser grained problem from the partial solutions is a mean to manage complexity. This divide and conquer principle can also be used as a basis for reuse (see below). A hierarchical decomposition / composition mechanism thus makes it possible to have different point of views on the modelled problem: coarse views that only show the coarser grained parts (aka black-boxes) or more detailed views that show finer-grained details inside coarser grained views (aka grey boxes).

Reuse to gain efficiency and quality. Reuse is the first step to mature engineering. It makes it possible to use previously designed problem solutions thus making it faster, less expensive and more reliable to reach the solution to the new problem. Reliability is

increased because a reused component has been thoroughly designed by experts (which might not be the case for a freshly designed one) and is supposedly more robust as it has been used (tested) by several people and in several contexts already.

Evolve to correct faults and take new user requirements into account. As an ability and not a quality, maintainability is not easy to measure [73, 34]. It nonetheless is central for software engineering. Easier maintenance indeed decreases the costs of software evolution, either it be to meet new requirements (that come from the user or from a changing environment), increase software quality or correct some detected failure. Software evolution is a crucial phase of software engineering as it is a life-long process that concentrates most of the costs and conditions software usability as times goes by. This is attested by Lehman's laws of program evolution as for example:

- *"A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful"* [69].
- *"Programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment"* [70].

1.2 Component-Based Software Engineering

Component-based software engineering (CBSE) is a software engineering paradigm that aims to provide better solutions to overcome these issues, as defended in Ivica Crnković *et al.* [38]: *"To achieve its primary goals of increased development efficiency and quality and decreased time to market, CBSE is built on the following four principles. Reusability [...]. Substitutability [...]. Extensibility [...]. Composability [...]."*. This can be reconciled to the above-mentioned main issues of software engineering if one considers component substitutability and software extensibility to be means to reach software evolvability.

1.2.1 Components are reusable assets

In order to achieve a better reuse of software, several abstractions have been proposed over time from libraries and modules to classes or services, from traits to mixins or aspects. Software components are one of them. In these different proposals or coding paradigms, the piece of software to reuse is of variable size and its role in the resulting software is of a variable nature. First, technical code, such as classical data structures, input / output capabilities or classical algorithms have been proposed as reusable assets in software libraries. This technical code indeed is highly reusable and does not present a lot of variability. Then, GUI components have been proposed in standard libraries. Following, reuse has been considered to encompass business code and several abstractions have been proposed as the reuse unit: modules, classes and objects, traits, components, etc. The point is to identify which one suits best. This depends on their size but also on

the nature of the available meta-information that is going to help (re)use them. The time where library contents had to be known by the architect or the developer as a preliminary knowledge is gone. To implement reuse at a larger scale, content to be reused is not known in advance and immutable. The reusable artifacts thus have to be documented in a certain way that makes their usability information accessible to humans, but also to automated programs.

Components are at the crossroads of many different research fields and thus can be given various definitions.

Research works on Architecture Description Languages (ADLs) [77] have long been talking about components. According to David Garlan *et al.*, components are said to "*represent the primary computational elements and datastore of a system*" [54] and be "*the locus of computation and state*" [99] as Mary Shaw *et al.* points out.

In UML 2.5 [89], "*a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required Interfaces. A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces.*"

According to the very famous definition of Clemens Szyperski's book [107], a software component is "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*".

Heineman and Councill [63] define a software component as "*a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*".

Components are typically described as grey boxes.

Unlike modules and classes which solely provide a description of the services / functionalities that are provided by the module, software components try to avoid the possible side-effects of hidden dependencies. To do so, their external view exposes both the services they provide (their capabilities) and the services they require (their needs). This is the first asset of software components that is going to make it possible to assemble them by comparing their types as these types describe the two points of view of their potential collaboration, both the server and the client side. Components can thus be considered as fully packaged pieces of code, ready to be reused if assembled together and deployed in a component execution environment. There is no need to know their implementation in order to assemble them. This is why they are considered reusable.

Component assembly is a form of composition, sometimes called horizontal composition. Components' internal view can be hidden for decoupling.

When components are hierarchically composed of components which is the case in hierarchical (or composite) component models, their internal view (or inner architecture) itself is a component assembly. The relation that links a (coarser-grained) component to its inner (finer-grained) components is another form of composition, sometimes called vertical composition. Hierarchical component models are preferred as they enable both abstraction and complexity management. Hierarchical component models are also a cornerstone of component reuse as they encapsulate some existing component assembly into a component and in turn assemble it, at a higher granularity level, with other components.

Components must be designed thoroughly to be both versatile and usable through their parameterization. They must be sufficiently independent to be usable in various contexts (and not necessitate fixed extra material to be able to function). They also must have an adapted granularity as one will prefer a component that best fits his requirements without to much extra capabilities (that add unneeded dependencies which also means extra complexity for the architect and error sources). This is why a component repository must contain components of diverse granularities and that we recommend hierarchically composeable components, so as to easily build coarser grained components from small ones and be able, in turn, to reuse them.

1.2.2 Component reuse impacts the engineering process

Traditional software development processes have to be adapted to component reuse [37, 32]. As compared to traditional development processes, CBSE presents the advantage of separating concerns for developers and architects. **Component engineering** follows a traditional development scheme[1] and results in components stored in a repository and available for reuse from there (see left part of Figure 1.1).

After component engineering comes the second step: the engineering of software applications from components. **Software development by component reuse** is about finding into dedicated repositories the components that match the application requirements and assemble them into applications (see right part of Figure 1.1). This process is clearly separated from the previous one. It neither requires the same languages and tools nor the same capabilities. Moreover, provided that the component interaction constraints are clearly expressed, this separation does not require to know the technology (language, communication protocol) used to code a component.

[1] This scheme is traditional in that it is one of the main concerns of software engineering research and practice. Development can follow whatever process (waterfall, V, spiral, agile, etc.). It is not the central concern of our work but the interested reader can refer to, for example, Chapter 3 of Van Vliet's book [112] to have an overview.

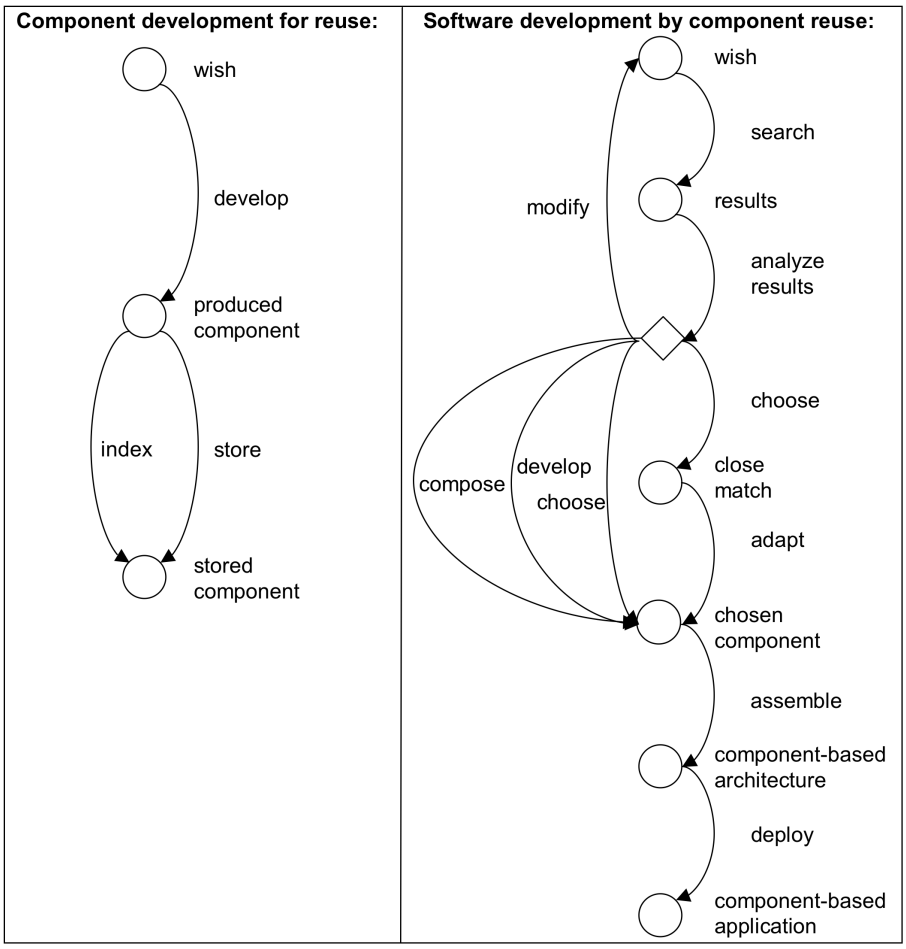


Figure 1.1: Activities of component development for reuse and of software development by component reuse

As shown on the right part of Figure 1.1, the activities for searching, choosing, adapting and assembling reusable components from repositories are very different from the activities of component development for reuse. First, the requirements must be expressed (as wished) in the form of an hypothetic component that will be searched for. Depending on the results of the search, several activities follow:

- if a single matching component is found, then it will be used in the software architecture and assembled to the others found to do so.

- if too much components match the query, the component that is going to be used has to be chosen among those. The choice can be anything from random to any combination of strengthening constraints (for example on qualities, usage, ranking, etc.) available and suitable that still provide solutions as a subset of the initially found set of possible solutions.

- if no component is found, there are several possibilities:

- ◇ the wished component can be built by assembling smaller grained components from the repository. Such a composition might result in a composite that exactly matches the needs or by a component that closely matches the needs and must further be adapted.
- ◇ another relaxed search might be performed that searches for a component that matches the needs as closely as possible. Missing characteristics can the further be automatically or manually completed, depending on their complexity (component adaptation).
- ◇ the wished component can be developed from scratch.

Once components are chosen, they are assembled to form a component-based software architecture then deployed into a component-based software.

1.3 Research questions

This development process being set, several research questions arise.

- ○ q1 : how has the component repository to be indexed so as to ease component search?
- ○ q2 : how is the wished architecture going to be expressed?
- ○ q3 : when considering a component's interface, which other interface can it be connected to?
- ○ q4 : what are the interfaces of a given component that have to be connected?
- ○ q5 : when is a component assembly correctly (fully) connected?
- ○ q6 : can components be assembled automatically?
- ○ q7 : what parts of software can evolve? its code? its design model? its specification?
- ○ q8 : do evolutions need to be anticipated?
- ○ q9 : can evolution be automated?

These questions are going to be tackled by the various contributions of Chapter 2. They all target a specific activity or aim at a specific result from the activities of component development for reuse and of software development by component reuse as synthetized graphically on Figure 1.2.

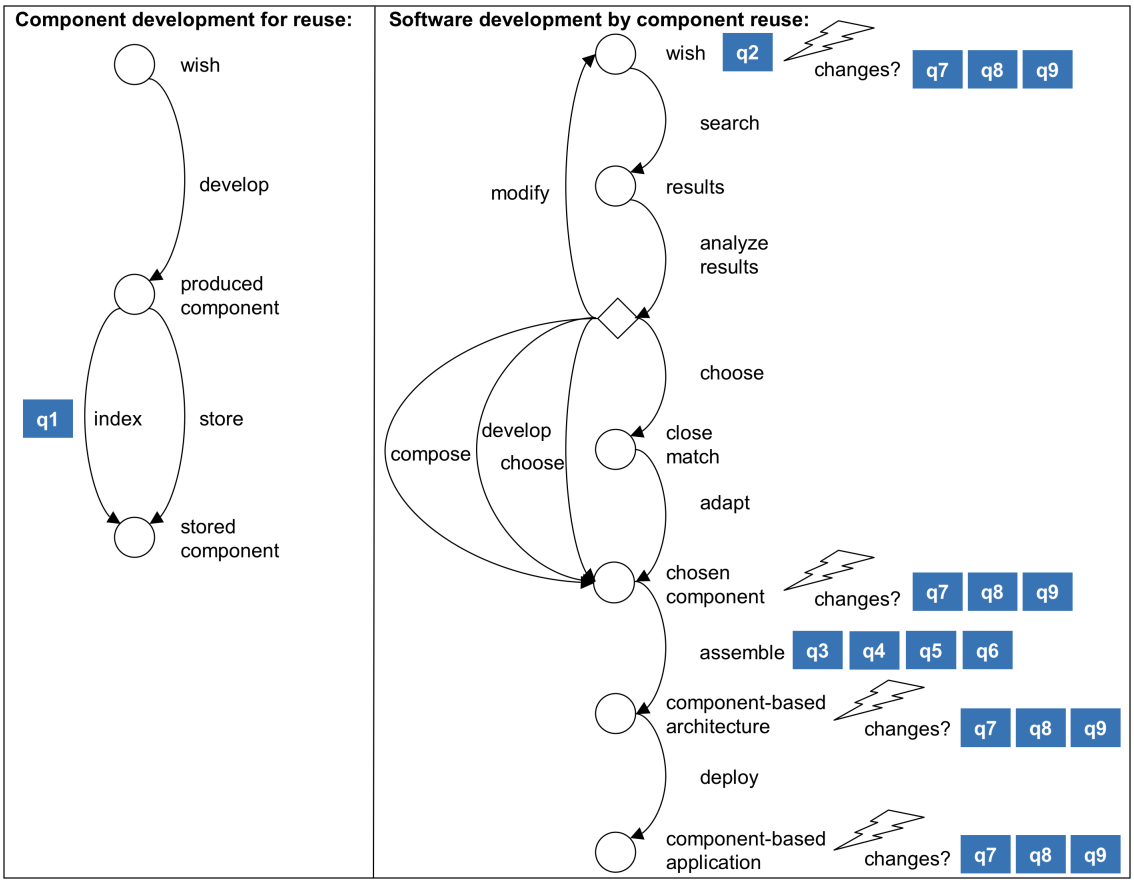


Figure 1.2: Research questions and their corresponding activities / results of CBSE

1.4 Organization of the document

This habilitation thesis comprises two main chapters.

Chapter 2 presents a synthesis of my contributions on composition, reuse and evolution in component-based software engineering. A first section briefly introduces the formal models of components I have proposed. Next section presents my work on indexing components into repositories. A third section introduces the Dedal ADL that was proposed to support component reuse. After that, a fourth section is about architecture composition by assembling port-enhanced components. Last section spans several contributions on software architecture evolution.

Chapter 3, in turn, concludes this dissertation. A first section is dedicated to a synthesis. Then, I briefly introduce the chosen appended articles before drawing some perspectives to my research.

Appendices follow the bibliography and compile five research articles – four of which published in journals, one presented at a conference – that complement the dissertation.

"L'intelligence est caractérisée par la puissance indéfinie de décomposer selon n'importe quelle loi et de recomposer en n'importe quel système."

Henri Bergson (1859 – 1941),
L'évolution créatrice.

Chapter 2

Synthesis of contributions

This chapter is going to present the subjects I have been working on in the field of component-based software engineering.

After briefly introducing the need for a formalized model of components, I will present how I used Formal Concept Analysis as a classification technique to order components according to a component substitutability relation, thus proposing a pre-calculated index of components into repositories that will ease later searches.

Then, I will show how adding a third specification level into and ADL is a basis for a better support of component reuse. I will present Dedal, such a three-level ADL tailored for component reuse.

After this, I will present how enhancing components with ports that group together interfaces and define a more precise connection semantics can provide a means to calculate whether components are connected in a manner that guarantees that the assembly supports the desired functionalities. This is a means to connect only what is necessary thus supporting reuse and avoiding unnecessary causes of errors.

To conclude, I will present several viewpoints on managing software architecture evolution, from co-versioning to managing changes as first class entities and calculating evolution plans.

2.1 Definition of the component model

Chapter 1 has shown that components are reusable pieces of software. Their reusability is possible thanks to the fact that the external view of these components explicitly exposes their dependencies to their environment, that is to other components they can be assembled to compose a software system.

Several differences exist between component models. As there is no widely accepted

standard or consensus, working on component-based software engineering starts with defining the component model that is going to be used.

There are common acceptions though. Most of the models agree on the fact that the external description of components contains their declared provided and required interfaces that are each composed of a set of functionality signatures.

There is less consensus on ports however (see Section 2.4). The fact that component models are hierarchic is not widely accepted but for most hierarchical component models, the inner view of components is a component assembly where relations are of two types: connections between opposite direction interfaces of distinct components and delegation between same direction interfaces one of which is from the inner assembly, the second of which is of the composite (*i.e.*, belongs to the external view of the hierarchically coarser component).

In order to be able to reason about components, I have proposed several means to model them, each corresponding to a different presented contribution:

- with the dedicated (informal) modular Dedal ADL [119, 117, 121].

  In this component model, defined during the PhD thesis of Huaxi (Yulin) Zhang, components are described textually, using the concrete syntax of our proposed Dedal ADL (see Section 2.3). The component model is quite minimal but nonetheless hierarchical.

- with a formal set-theory inspired *ad hoc* notation [43, 42].

  This component model, defined during the PhD thesis of Nicolas Desnos, is described mathematically. This makes it possible to define a precise semantics. The component model is hierarchic and extended with the notion of ports (primitive and composite) as presented in Section 2.4. The formal nature of the model helped precisely define how components connect to one another (research question q3) and when a connected component assembly is completely connected (research questions q4 and q5). This last property calculates for components and generalizes to assemblies. It depends on the novel connection semantics given to ports. The calculability of this property further makes it possible to automatically build component assemblies (see Section 2.4).

- with the formal and tooled B language [84, 81].

  This component model, defined during the PhD thesis of Abderrahman Mokni, is called formal Dedal. It is the formalized form of the model of the Dedal language (see Section 2.3). Expressed in B [3], a first order set-theoretic formalism, this model has the advantages of being both mathematically founded and tooled with solvers. Well-formedness of instances of this model can thus be automatically

checked. The validity of formalized properties can also be automatically assessed. The solver is even used as a planning tool to search for sequences of actions that restore a given property (see Section 2.5.4).

2.2 Indexing components into repositories using Formal Concept Analysis

In order to be able to define operations that involve several components such as component connection or component substitution, there is a need to type components. It then becomes possible to define various comparison relations between those types. In my work, the subject of component typing thus is a foundation subject. I have explored several ways to define and compare component types.

2.2.1 Issues of components typing

Typing components is a novel idea. A lot of research has been led, however, on functionality substitutability and on ordering types in object-oriented programming languages. Indeed, these mechanisms are the theory on which compiling techniques have been built upon and have been explored quite extensively by early works on class inheritance. Typing has also been explored from the point of view of analyzing the dynamics of systems. In this case, the dynamics can be described using various formalisms such as: state-charts [62], state-machines (*e.g.*, UML behavior state-machines or protocol state-machines [89]), Petri-nets [87], etc. At last, typing can also be inferred from more semantical information such as natural language descriptions that can be found in meta-data descriptions such as WSDL [113] descriptiors.

Information type classification axes. It is a fact that component typing can be founded on several sorts of information. The types of this information can be classified using three criteria.

- the IE criteria. Some information are *intrinsic (I)* to code. They do not necessitate further human intervention. Others are *extrinsic (E)* meta-data, that is a part of the documentation that accompanies code.This information can be missing or be badly written. If not, it can be very rich and possibly more difficult to analyze automatically.

- the SD criteria. Some of the information is *static (S)*. The other documents the *dynamics (D)* of the component behavior.

- the SySe criteria Some information is said to be *Syntactic (Sy)*. The other documents the *semantics (Se)*.

Nature of the information. The information itself can be of different nature. We can list:

- n1: functionality name,
- n2: functionality input parameter types,
- n3: functionality input parameter names,
- n4: functionality input parameter number,
- n5: functionality input parameter order,
- n6: functionality output parameter type,
- n7: exception types that can be thrown by the functionality,
- n8: interface (or service) name,
- n9: interface (or service) types (as sets of functionality signatures),
- n10: list of interfaces that are provided by a component,
- n11: list of interfaces that are required by a component,
- n12: textual documentation on interfaces,
- n13: behavior protocols attached to interfaces,
- n14: global behavior protocol attached to a component.

Each nature of information can relate to one or more information types.

Functionality substitutability based on their static types. In order to compare functionalities, a technique is to compare their signatures in terms of their domain (the cartesian products of their parameter types) and range (their output parameter types).

Parameter types can be partially ordered by their substitutability relations. These have been theoretized by Barbara Liskov's substitution principle [74, 75]. The **substitution principle**, as defined in those works, relies on a partial order based on object (static) types that can be explained as is: if a type *Sub* is a subtype of another type *Typ* then, whenever an object of type *Typ* is expected in a program, it can be substituted for an object of type *Sub* without altering the program.

This rule generalizes to function signature comparison. For a function f_{sub} to be a valid substitute of a function f, their parameters must obey the following rules:

- there must exist a matching between the parameter types of fonction *f* and those in f_{sub} such as each parameter type *Sub* among *f*'s have a matching parameter type *Typ* in f_{sub} such that *Sub* is a subtype of *Typ*. Functions are often said to have **contravariant input parameter types**.

- the output parameter type of f_{sub} must be a subtype of that of *f*. Functions are often said to have **covariant return types**.

This well known substitution principle uses information with a limited number of natures. They correspond to information natures number n2 and n6, according to the above enumeration.

2.2.2 Formal Concept Analysis basics

Formal Concept Analysis (FCA) is a data mining and classification theoretical tool [26, 53]. It relies on various mathematical structures among which are concept lattices that represent partially ordered set of concepts mined from a dataset composed of objects described by attributes.

A **formal context** is a triple $K = (O, A, R)$ where O and A respectively are object and attribute sets and $R \subseteq O \times A$ a binary relation. Table 2.1 gives the example of a formal context that models information about vegetables, their edible part and their botanical families.

A **formal concept** is a pair (E, I) composed of:

- an object set, the **extent**, subset of O, composed of all the objects that share all attributes in I and defined by $E = \{o \in O | \forall a \in I, (o, a) \in R\}$,

- an attribute set, the **intent**, subset of A, composed of the all attributes that are shared by all elements in E and defined as $I = \{a \in A | \forall o \in E, (o, a) \in R\}$.

Given a formal context $K = (O, A, R)$ and two formal concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$ of K, the **concept specialization partial order** $\leq_s$ is defined by $C_1 \leq_s C_2$ if and only if $E_1 \subseteq E_2$ (extents covary). It could be defined equivalently by $I_2 \subseteq I_1$ (intents contravary). C_1 is said to be a sub-concept of C_2 and C_2 a super-concept of C_1.

The set $\mathcal{C}_K$ of all concepts of a formal context K provided with the $\leq_s$ specialization partial order is the **concept lattice** $(\mathcal{C}_K, \leq_s)$ associated with K.

For readability's sake, diagrams (see Figure 2.1) often present extents (white labels) and intents (gray labels) in a simplified way such that each object or attribute appear only once. This amounts to removing top-down inherited attributes and bottom-up included objects.

	Root	Stem	Leaf	Flower	Fruit	Seed	Asteraceae	Fabaceae	Solanaceae	Cucurbitaceae	Cruciferae	Chenopodiaceae	Alliaceae	Apiaceae	Poaceae
Carrot	×													×	
Beet	×		×									×			
Radish	×		×								×				
Celery		×												×	
Broccoli		×		×							×				
Lettuce			×				×								
Spinach			×									×			
Cauliflower				×							×				
Artichoke				×			×								
Tomato					×				×						
Cucumber					×					×					
Eggplant					×				×						
Zucchini				×	×					×					
Corn						×									×
Bean						×		×							
Pea						×		×							
Onion	×												×		

Table 2.1: The formal context of vegetables

In order to scale up, we sometimes consider the Attribute Object Concept poset (AOC-poset) instead of the lattice. The **AOC-poset** is the sub-order of $(\mathcal{C}_K, \leq_s)$ restricted to object-concepts (concepts the simplified extent of which are not empty) and attribute-concepts (concepts the simplified intent of which are not empty). As such, AOC-posets do not remove any significant information and scale much better than lattices.

Figure 2.1 gives an example of concept lattice. It is generated[1] from the above formal context of vegetables. One can see on the lattice that we eat the leaves of all known sorts of chenopodiaceae (spinach and beets). One can also see that, among known cucurbitaceae, cucumbers are the only species from which we only eat the fruit. Indeed, the other cucurbitaceae we know, zucchinis, have edible fruits and flowers.

2.2.3 Components classification based on their syntactic types

Starting from functionality signature substitutability, this notion of substitutability is first extended to interface (sets of functionality signatures) types and further to component

[1] To do so, the *conexp* concept explorer open-source tool (`http://conexp.sourceforge.net`) was used.

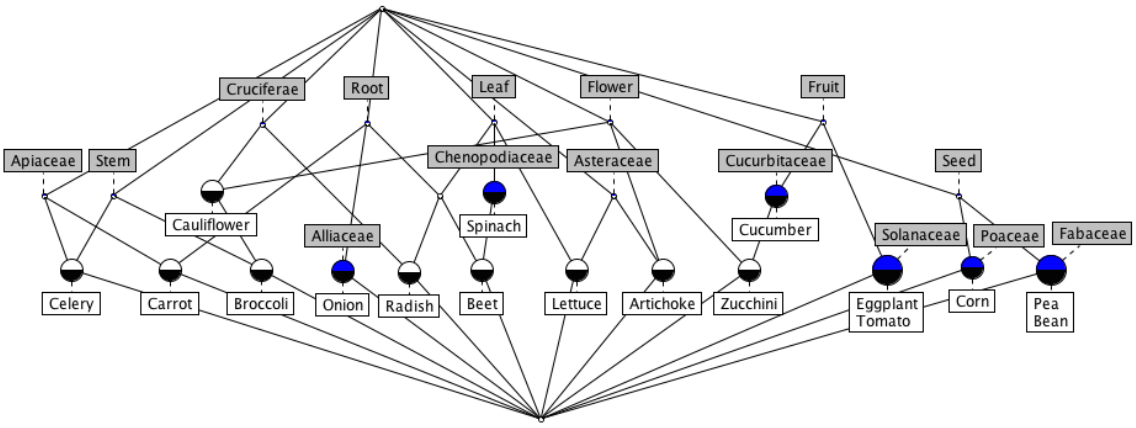


Figure 2.1: The vegetable concept lattice

(sets of interfaces) types. The need appears for a classification technique that can produce a partial order on groups of characteristics (groups of parameter types, groups of functionality signatures or groups of interfaces). Formal Concept Analysis is chosen as this technique because it creates and classifies formal concepts as demanded but also because it mines new abstract concepts that are created for the purpose of classification. We will explain why this is interesting at the end of this section. This work has been the opportunity of an international collaboration with Gabriela Arévalo from Argentina [16, 17, 18, 2, 20, 19, 1].

Statically typing components provided and required functionalities. In order to provide types to components and thus be able to define component substitutability or component compatibility, one of my contributions is a generalization of such substitution principle to components. The result is a definition of component types that relies on Barbara Liskov's substitutability principle and extends it to fit the needs of components.

First, a slightly different definition of (provided) functionality substitutability was considered.

We indeed added a naming constraint on functionalities, choosing not to concentrate solely on typing as in Barbara Liskov's work but define a substitutability mechanism that resembles those of programming languages where only homonymic functionalities can be substituted to one another. We also considered that the substitute functionality can declare the need for less input parameters than the functionality it replaces.

We then transposed the definition of substitutability from provided functionalities to required functionalities. As these latter are the opposite of the first, all constraints are reversed.

For a required function f_{sub} to be a valid substitute of another required function f,

they must obey the following rules:

- f_{sub} and f must have identical names,
- input parameter types must be **covariant**,
- return types must be **contravariant**.

The proposed component classification process. The proposed component typing process is divided into three steps, each of which uses FCA as a classification mechanism. The entry of the process is composed of type hierarchies (so as to be able to compare functionality parameter types). Then, the process recursively classifies:

- **functionality signatures**. Homonymic functionalities are classified based on their full signatures using FCA and the type hierarchy of their input and output parameters according to the substitutable type inference rules mentioned above.
- **interfaces**. They are classified based on their types (as sets of functionality signatures; their name does not matter) using FCA and the classification of their functionality signatures as pre-calculated substitutable functionality inference rules. Provided interfaces and required interfaces are dealt with separately.
- **components**. They are classified based on their types (as sets of provided and required interfaces; their name does not matter) using FCA and the classification of both their provided and required interfaces as pre-calculated substitutable interface inference rules.

Figure 2.2 provides an overview of the three steps classification process.

On the top left part of the figure, the type hierarchy is the input information on object types. It is composed of two connected components: one for the `Location` and the other for the `Route` groups of types. This object type partial ordering serves to classify functionality signatures. On the top right part of the figure, the `route` functionalities are classified according to their parameter types (here, only their output types vary) and an extract of the lattice is shown. This functionality signature partial ordering in turn serves as an entry to classify interfaces. In the bottom right of the figure, provided `Route` interfaces are classified and an extract of the lattice is shown. Here, the only variation is the signature of the `route` functionality. In a last step, the interface partial ordering serves as an entry of the component classification process. On the bottom left part of the figure, `RouteCalculation` components are classified and an extract of this classification shown. One of the variations between `C4` (`BotanicRouteCalculation`) and `C10` (`TouristicRouteCalculation`) is the provided interface at their bottom right corner which are the ones that were classified in the shown extract of the previous step.

The result of this classification process is a component type poset, the partial order being the component substitutability relation. This poset thus acts as an automatically

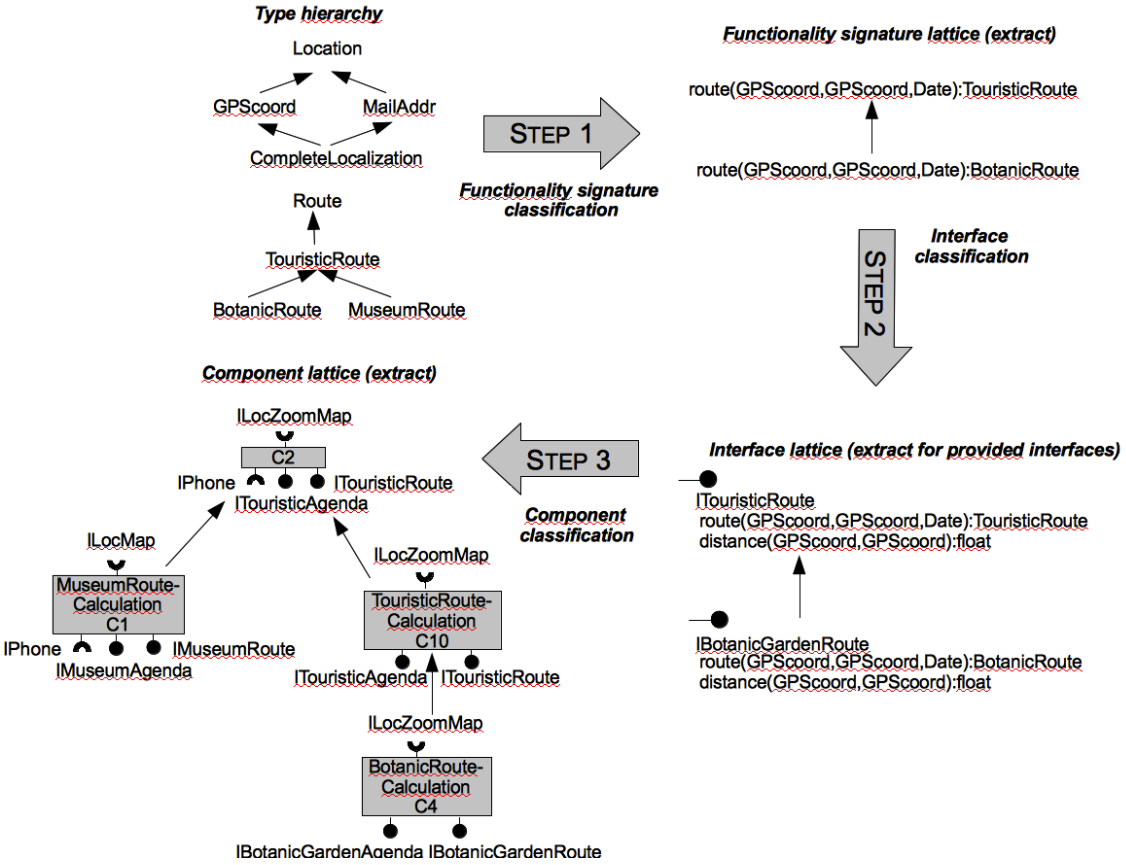


Figure 2.2: Synopsis of the three-step component classification process [1]

pre-calculated component index that eases the search for substitutable components and can therefore be a support for architecture evolution. It comprises both real components – the ones that served as data for the classification process – and abstract (mined) components – components that do not exist but are interesting generalizations of existing components. Abstract components are discovered abstractions that could be useful in a component reengineering process.

The same poset can also be used to assemble components (that is, compose architectures from components). To do so, the component candidate to composition must be mirrored (its interface directions reversed), optionally classified in the component hierarchy (if the mirrored component is not already there) and then possible substitutes searched for. As they are substitutable with the component that has the candidate component exact matching type (which is the most obviously connectable component type), the list of possible substitutes can all be connected to the candidate component.

The proposed approach that classifies components based on their syntactic types uses information with a limited number of natures to define component types. They

correspond to information natures number n1, n2, n6, n9, n10 and n11 according to the above enumeration.

This classification can be used as a pre-calculated index in order for component substitutes to be easily found. It is a possible response to research question q1.

2.2.4 Service classification based on semantical meta-information

During the PhD of Zeina Azmeh, classification was explored in several complementary way [23, 24, 22]. The objective was to classify web services and thus, the classification criteria were naturally influenced by the information retrieval research field.

Requirements are modelled in a way inspired from the abstract specification of the as-wished architecture in the CBSE process (see Figure 2.4). Firstly, an abstract WSDL [113] descriptor describes all the needed services by specifying functional and non-functional requirements. A needed service is characterized by the functionalities it provides (along with its input parameter names and types and its output parameter type) and the expected QoS levels for every supported quality attribute. Secondly, the messages that can be exchanged by such described web services (their orchestration) are described in an abstract BPEL [88] descriptor. An abstract BPEL descriptor differs from a (normal) concrete BPEL descriptor in that it refers to abstract services described in (a local) abstract WSDL descriptor, instead of concrete services retrieved from the web.

This system requirement model is somewhat similar to the specification of a component-based architecture, where abstract WSDL descriptors can be compared to the external view of components (their interfaces) and abstract BPEL descriptors to the abstract architecture. Abstract web services only describe their provided functionalities and abstract BPEL descriptors describe the dynamics of the service-based composition (aka orchestration) instead of focusing essentially on the structure in CBSE.

As it is the case in CBSE (see Section 2.3), the information conveyed in these descriptors serves as a guide to search for concrete services that match the wished ones. To do so, an index of similar substitutable services is built by the proposed service classification process [22] which decomposes in two phases and uses FCA at each of them two.

Phase 1: Finding groups of similar functionalities. In this work, web-services are searched for in various data sources on the web using search engines. They are described by their WSDL descriptor which acts as external meta-information on its functionalities. The information available is both syntactical (functionality signatures) and semantical (free natural language text). As they come from various sources and they have been developed and documented separately, one cannot expect that web-services be named, or manipulate parameter types that are named, in an identical manner.

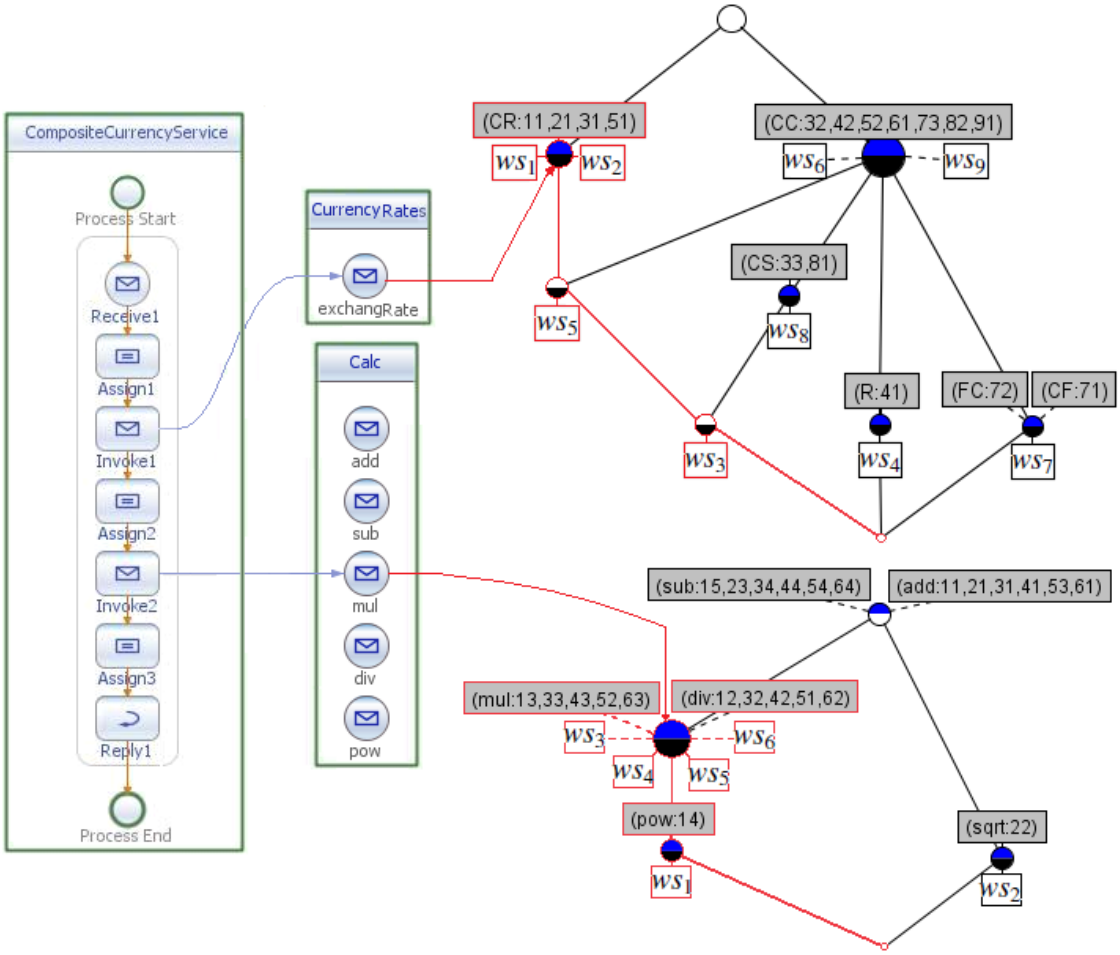


Figure 2.3: The composite currency service, supported by backups from the service lattices [22]

Though, searching for web services amounts in being able to identify close services or, close functionalities by semantically comparing these names and / or the types. This can be done in various ways such as vector space modelling inspired techniques [115, 93], clustering using machine learning techniques [64, 36] or ontology-based techniques [27].

In this work, it is done by comparing, as two natural language strings, all pairs of functionality signatures to measure their similarity [48, 106, 68] and then clustering them using FCA techniques. The chosen similarity measure for experiments is the Jaro-Winkler distance [116]. It provides a value between 0 (incomparable functionalities that are served by a same web-service) and 1 (strict equality). If we consider the relation between the set of functionality signatures and itself (where the object set equals the attribute set), this leads to a multi-valued (non binary) formal context that is a square similarity matrix of diagonal 1 which presents the property of being symmetrical (because of the symmetry of the similarity measure). If the measure values are applied a threshold

to make them binary, the matrix then becomes a symmetrical formal context. Applying FCA therefore results in a lattice where similar functionality (according to the chosen measure and threshold) are grouped into formal concepts.

Phase 2: Using these groups to classify services. Using these functionality groups as if their member functionalities were a single one, the containment services $\times$ group relation is modelled as a formal context. Applying FCA results in a service lattice that can be used to find similar services (grouped in same formal concepts) or substitutable services (linked by a sub-concept to super-concept relation).

Figure 2.3 shows an example of a composite service that is described by a BEPL process involving several sub-services that themselves refer to the lattice of their classified similar services that could be used as backups.

The proposed approach that classifies services based on semantical meta-information uses information with a limited number of natures to define service types. They correspond to information natures number n1, n3 and n4 according to the above enumeration.

The approach constitutes a variation of the indexation method proposed in Section 2.2.3 tailored for web-service, so as possible substitutes to web services can be easily found. It is another possible response to research question q1.

2.3 Dedal, a three level ADL that supports component reuse in CBSE

2.3.1 Issues of architecture-driven component reuse

Building software applications from components amounts to have models of components and models of applications described in a non-procedural language. Architecture description languages (ADLs) which are used to do so are declarative domain specific languages (DSLs). Some of them are XML dialects. Others have their own concrete syntax. UML diagrams (or slightly modified UML with profiles) can even be used to do so.

There is a close analogy with model-driven engineering. Indeed, the grammar of ADLs can be described as a meta-model, their abstract syntax as both an instance of the meta-model and a class model for concrete architecture definitions (that can be seen as instances of the model that describes the syntax).

Therefore, there is a convergence in the tools used to define languages and models, especially accompanying UML-based tools. Indeed, the Eclipse IDE and its MOF are tools that can be used to easily describe the abstract and concrete syntax of an ADL (see Section 3.3.4).

This analogy can be continued further to describe the continuum, in terms of models of a software, from its specification to its design, its implementation and, furthermore, to its runtime. As models are everywhere, even at runtime (Models@runtime) they can provide at any development stage the necessary information to make design decisions or to discipline changes that can occur on software.

This is why, designing an ADL to support reuse, it must represent the various points of views on the software system that correspond to each development stage and that are available at any time so as to guide component choices (reuse) and software evolution (as described in Section 2.5).

Apart from this characteristic, the qualities of an ADL are those of a component model: modular (so as to be able to reuse small fragments), loosely coupled (so as to distinguish types and their implementations) and hierarchical (so as to manage complexity and favor reuse).

2.3.2 The three levels of Dedal

The process of software application development from component reuse has three main development steps: specification (the `wish` product on Figure 2.4), implementation (the `component-based architecture` product on Figure 2.4) and deployment (the `component-based application` product on Figure 2.4).

The Dedal architectural model has been proposed as one of the contributions of the PhD of Huaxi (Yulin) Zhang [119, 117, 121]. It models the views of the architecture at each of these three development steps:

- **Model of requirements.** The `model of the wish` product of Figure 2.4 models the requirements. The as-wished architecture description is represented as an abstract architecture specification in Dedal where abstract component roles model the ideal components the architect would like to reuse.
- **Model of design.** The `component-based architecture` product of Figure 2.4 models the design.The architect creates as-found the concrete architecture configuration by reusing existing components classes from the repository.
- **Model of runtime.** The `component-based application` product on Figure 2.4 models the runtime software. Component instance assemblies describe the runtime constraints on instantiated components.

Figure 2.4 proposes the software application development from component reuse activities with their corresponding architectural models that are intended to guide reuse and evolution.

Figure 2.4 also shows that changes can occur at any of the description levels which is the more realistic option of modelling change occurrences because architects and developers hardly stick to a top-down approach. Such option however implies that both

forward (model-driven) and reverse (retro) propagation mechanisms be implemented so as to propagate the impact of changes to other models (see Section 2.5). This is an answer to research question q7 in the form of an *a priori* manifesto.

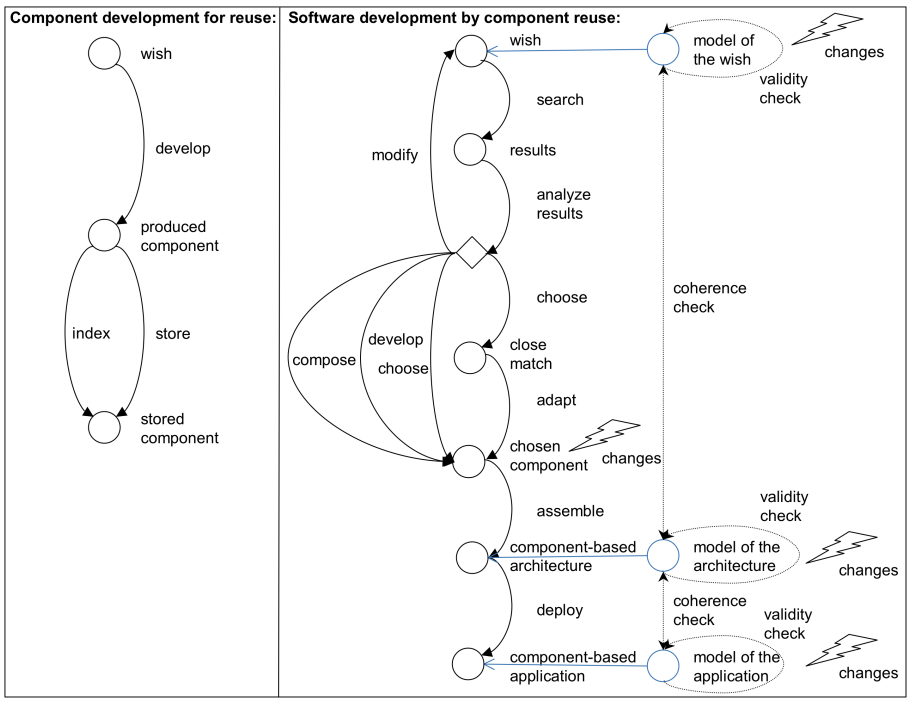


Figure 2.4: Activities of CBSE and their corresponding architectural models

The Dedal ADL defines variants of components and connectors at those three levels:

- **Component roles** model abstract component types in the **abstract architecture specification**.
- In the **concrete architecture configuration**, components are modelled by their **component types** and **component classes**. Component types define the reusable full types of at least one (maybe several) existing component implementations. They are defined by describing the interfaces and behavior of these component classes. Component classes describe concrete component implementations. Each component class implements a component type and each can either be primitive or composite.
- **Component instances** document the real artifacts that are connected together in an **instantiated component assembly** at runtime.

Figure 2.5, Figure 2.6 and Figure 2.7 provide small examples of (a part of) the concrete syntax of the Dedal ADL.

```
component_role BikeCourse
 required_interfaces BikeQS; CourseQS
 provided_interfaces BikeOprs; CourseOprs
 component_behavior
  (!BikeCourse.BikeOprs.selectBike,
  ?BikeCourse.BikeQS.findBike;)
  +
  (!BikeCourse.CourseOprs.startC,
  ?BikeCourse.CourseQS.findCourse;)
```

Figure 2.5: Dedal description of the BikeCourse component role

Figure 2.5 defines a component at the specification level (it is called a component role). It provides its name, its required and provided interface names (to refer to external descriptions of the interface types), a behavior protocol (written as a regular expression inspired from SOFA [92] where the ! and ? symbols respectively prefix emitted and received messages).

```
component_type BikeTripType
  required_interfaces BikeQS ; CourseQS; LocOprs
  provided_interfaces BikeOprs; CourseOprs
  component_behavior
    (?BikeTripType.BikeOprs.selectBike,
    ?BikeTripType.LocOprs.findStation,
    !BikeTripType.BikeQS.findBike;)
    +
    (?BikeTripType.CourseOprs.startC,
    !BikeTripType.CourseQS.findCourse;)
```

Figure 2.6: Dedal description of the BikeTripType component type

Figure 2.6 defines a component type at the configuration level. It provides its name, its required and provided interface names (to refer to external descriptions of the interface types) and a behavior protocol (also written as a regular expression).

```
component_class BikeTrip
 implements BikeTripType
 using fr.ema.BikeTripImpl
 versionID 1.0
 attributes string company; string currency
```

Figure 2.7: Dedal description of the BikeTrip (primitive) component class

Figure 2.7 defines a component class at the configuration level. It provides its name,

the name of the component type it implements, a link to the concrete implementation class, a version ID and a list of observable attributes (that are used to describe constraints at the assembly level).

The link between the component class and the component role it realizes is not described here but in the description of the configuration in the form:

```
BikeTrip (1.0) as BikeCourse;
```

2.3.3 Inter-level relations in Dedal

A special focus is put on inter-level relations that define how a concrete architecture configuration is compatible with its abstract architecture specification and how an instantiated component assembly corresponds to its concrete architecture configuration. Indeed, these relations vehicle an important part of semantics of the three level architecture level model as they define whether the three descriptions are coherent with one another. These inter-level relations between descriptions are declined on their constituents. As most of the semantics is vehicled by components in Dedal (as for now, connectors are passive semantics-free links), the inter-level relations the semantics of which we are going to focus on are those between component descriptions in the three levels.

Figure 2.8 and Figure 2.9 represent the relations that are set between the different representations of a given component. A component role can be realized by several component classes (each being a different concrete realization of the specification carried by the role). A given component role can be partially realized by a component class, thus necessitating several component classes (each realizing a part of it) to fully realize the role. Each component class implements a single component type which can be implemented by any number of component classes (each fully implementing its type). A component type matches a component role either fully or partially (as for component classes). A component instance instantiates a single component class. A given component class might have several instances.

The Dedal ADL and these inter-level relations have been formalized using the B language [3]. The result is a new formal ADL expressed in B: formal Dedal. This has been done during the PhD of Abderrahman Mokni [80, 84, 81].

Dedal architecture descriptions can be automatically transformed, in a model-driven engineering (MDE) approach, into formal Dedal descriptions. The formal ground of the B language together with the existence of B solvers make it possible to automatically verify that an architecture description is both well-formed (each level description is well-formed) and coherent (all inter-level relations are verified).

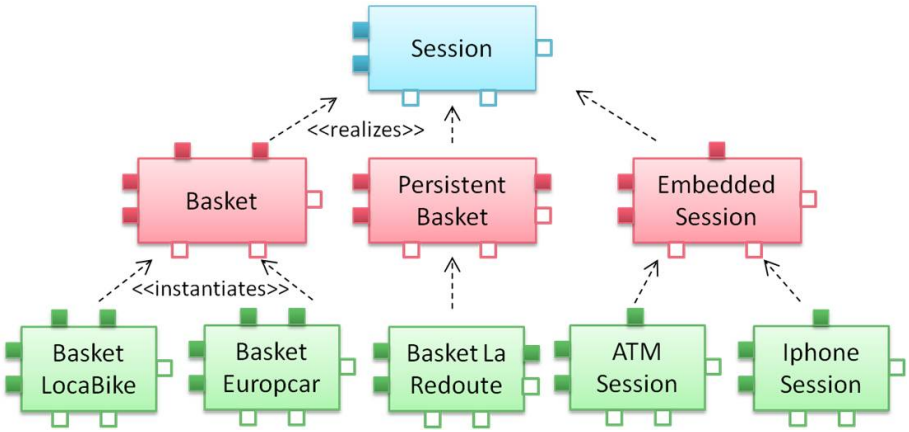


Figure 2.8: Inter-level relations between component descriptions (a) [117]

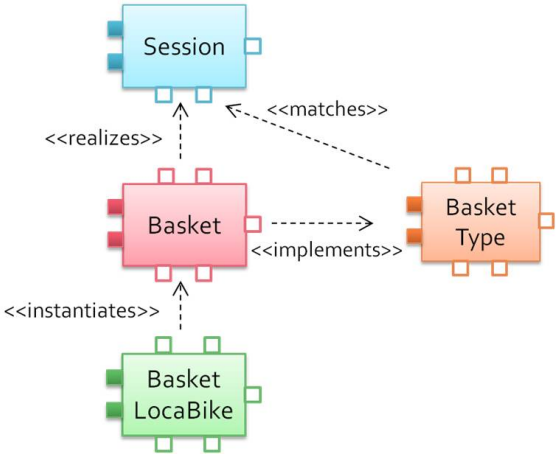


Figure 2.9: Inter-level relations between component descriptions (b) [117]

As an illustrative example, Table 2.2 shows the B machine that describes the configuration level.

There are three inter-level coherence rules. An example is the relation between a *CT* component type at the configuration level and its corresponding *CR* role at the specification level. It can be ruled as follows. A component type *CT* matches a component role *CR* iff an injection *inj* between the set of interfaces of *CR* and the set of interfaces of *CT* exists such that *int* can be substituted for *inj*(*int*), *int* being an interface of *CR*.

MACHINE $Arch\_configuration$
INCLUDES $Arch\_concepts, Arch\_specification$
SETS
COMP_CLASS; *CLASS_NAME*; *ATTRIBUTES*; *ATT_NAMES* ; *CONFIGURATIONS*
CONSTANTS
COMP_TYPES
PROPERTIES
/* Component types are also a specialization of components distinct from roles */
COMP_TYPES $\subseteq$ *COMPS* $\wedge$ *COMP_TYPES* $=$ *COMPS* - *COMP_ROLES*
VARIABLES
config, *config_components*, *config_connections*, *compType*, *compClass*,
class_name, *class_attributes*, *compositeComp*, *delegatedInterface*, *delegation*, ...
INVARIANT
compType $\subseteq$ *COMP_TYPES* $\wedge$
/* A component class has a name and a set of attributes */
compClass $\subseteq$ *COMP_CLASS* $\wedge$ *class_name* $\in$ *compClass* $\rightarrow$ *CLASS_NAME* $\wedge$
attribute $\subseteq$ *ATTRIBUTES* $\wedge$ *class_attributes* $\in$ *compClass* $\rightarrow \mathcal{P}$(*attribute*) $\wedge$
/* A composite component is also a configuration as it is constituted of component classes */
compositeComp $\subseteq$ *compClass* $\wedge$ *composite_uses* $\in$ *compositeComp* $\rightarrow$ *config* $\wedge$
/* A delegation is a mapping between a delegated interface and its corresponding one */
delegatedInterface $\subset$ *interface* $\wedge$
delegation $\in$ *delegatedInterface* $\rightarrowtail$ *interface* $\wedge$
/* A configuration is a set of component classes */
config $\subseteq$ *CONFIGURATIONS* $\wedge$
config_components $\in$ *config* $\rightarrow \mathcal{P}_1$(*compClass*) $\wedge$
config_connections $\in$ *config* $\rightarrow \mathcal{P}$(*connection*)

Table 2.2: Formal specification of the configuration level [81]

Formally [81]:

$$
\begin{array}{l}
matches \in compType \leftrightarrow compRole \wedge \\
\forall (CT, CR).(CT \in comType \wedge CR \in compRole \\
\Rightarrow \\
((CT, CR) \in matches \\
\quad \Leftrightarrow \\
\exists (inj).(inj \in comp\_interfaces(CR) \overset{inj.}{\rightarrow} comp\_interfaces(CT) \wedge \\
\quad \forall (int).(int \in interface \\
\quad\quad \Rightarrow \\
\quad inj(int) \in int\_substitution[\{int\}]) \wedge \\
)))
\end{array}
$$

The work on Dedal is an answer to research question q2: how is the wished architecture going to be expressed?

Once this as-wished architecture (abstract architecture specification) is modelled, there is a support of the modelling intentions of the architect that is going to be preserved in all stages of development and runtime. This model of the intentions serves as a guide to search for concrete components in the architecture design step. As such, it enhances reusability. It is also a guide for architecture evolution (see Section 2.5) as it keeps track of the architect's intentions to which thought changes can be compared to.

2.4 Architecture composition by port-enhanced components assembling

Assembling components is a central issue of component-based software engineering, whether it be at the specification level where assembling amounts to find roles and connect them though their interfaces so as to build a coherent architecture specification, at design-time where assembling amounts to select component classes of adequate types and connect their interfaces until the assembly is valid and forms a software configuration or at runtime where assembling amounts to connect component instances that verify the assembling constraints and can be followed by automatic deployment.

This section proposes my vision of component assembling, its issues and a proposed solution. This work has been led during the PhD of Nicolas Desnos [45, 44, 43, 42], some of which has been the opportunity to collaborate with Guy Tremblay from Canada.

2.4.1 Issues of components assembling

Assembling components amounts to connect them through their interfaces. A candidate provided interface of a component is going to be connected to a candidate required interface of another component so that the needs of a component in terms of capabilities are satisfied by another. The type of the candidate provided interface must be compatible with the type of the candidate required interface. Type compatibility can be interpreted very differently from a component model to another. To my sense, it is to be interpreted in a broader sense than strict equality, as described in Section 2.2. Finding an interface of opposite direction (required or provided) and of compatible type is an answer to research question q3: when considering a component's interface, which other interface can it be connected to?

This question is under-documented in state-of-the-art works. Most of them consider type equality as a sound connection criteria between interfaces. Such a connection scheme obviously over-constrains connections as sub-typing (which is a highly documented topic in object-oriented programming) is ignored. This is why we studied how objected-oriented sub-typing could be generalized to components in Section 2.2. In such a component matching process, there are two main differences as compared to objects. First, the required point of view, which did not exist in object orientation has to be taken care of. Typing rules thus have to be generalized to this new point of view on interfaces.

Secondly, as we are not aiming at substitutability soundness as a compiler would be (components can further be adapted either automatically or manually), the strength of the desired relation can vary. This can lead to multiple compatibility or substitutability relations as evoked in Section 2.2.

The other research question that has to be answered is question q4: what are the interfaces of a given component that have to be connected? The answer to this question is going to provide means to automatically decide wether a component is correctly connected to be able to function in an assembly. As a consequence, this property will also be computable for a whole component assembly (an architecture), thus providing means to verify the correctness of assemblies in an automated way while automatically assembling components or automatically repairing damaged assemblies.

In most component models, all the required interfaces of the used components are to be connected [114]. Automatically verifying such a property is trivial but this connection scheme over-constrains the software architecture: as components are likely to provide more services than strictly needed, fully connecting components implies searching for components that are never going to be used by the application. Such a search diminishes the chances to find reusable components that suit all needs (even if some of the needs are artificial). It makes architectures artificially large (thus error-prone) and furthermore diminishes the architecture's extendability (as artificial requirements are fulfilled in a manner that might be contradictory to future requirements).

In some models [111, 29], several interfaces are *a priori* declared as not mandatory to connect. Most of the times, components have more capabilities than strictly needed (because of them being reused) which means all required interfaces need not be connected in all contexts. These models consider architectures in which optional required interfaces are not connect as valid ones. To me, this solution fails to model what interfaces of components have to be connected as the optionality of services is not an intrinsic property of the component but probably depends on the context in which the component is used. Indeed, a service might be mandatory in some context and optional in some other context. Defining components with inherently optional required interfaces thus includes hidden assumptions on future component usage.

In a third category of models, dependencies between interfaces are dynamically calculated. When an interface is connected, all its dependant ones must also be. These models [39, 105, 55, 94, 95, 4] thus enable partial connection of components based on extra information on component behavior dynamics. Some are able to determine if an interface has to be connected or not by analyzing behavior protocols [92, 4], others by reasoning on interface automata [39]. Dimitra Giannakopoulou *et al.* [55] uses model-checking. Judith Stafford *et al.* [105] use dependency analysis.

The third category of models is the one that provides the best answer to connecting the smallest set of required interfaces possible while still satisfying all requirements. Such

connection policy, however, is expensive to compute. Comparing components' dynamics indeed requires complex algorithms. The idea behind this work was to be able to test compatibility of a chosen component with all the potentially compatible ones that are found in a component repository, having in mind the objective of exploring assembling possibilities automatically in order either to provide candidate assemblies for architects to choose from or even, in less critic environment such as home automation, try and automatically select an assembly to be deployed. For such an application, the need for a new connection criteria emerged. The criteria had to be more precise and context-aware than connections schemes of the two first categories but it also had to be less difficult to compute than dynamic comparison of component behavior [92]. The metaphor of the plug, that simultaneously connects several plots altogether to another compatible plug led to use the port notion as an adequate concept to gather dependencies between interfaces and have a more sophisticated component connection scheme to meet our needs.

2.4.2 Ports improve components assembling

Several component models introduce the concept of port but definitions vary. In most component models a port is an interaction point for components to interact with their environment. It is the case in ArchJava [13], Java/A [58] (which extends ArchJava with protocols), Wright [14] or in COMPO [104].

For other models, such as UML 2.5 [89] and ACCORD [109], ports group interfaces. Such ports can be seen as both interaction points and concepts that structure the external view of components by grouping interfaces that are likely to be used in a same component-to-component collaboration.

In the general case, a port is bi-directional (it is composed of both provided and required interfaces) as in UML 2.5 [89].

In most models, a port embodies a peer in a peer-to-peer collaboration. Peer-to-peer port connection then amounts to connect each element from a component's port to a compatible element from another single port belonging to another single component. UML [89] is, to my knowledge, unique in defining multi-peer connections between ports (a port from a component can connect to two distinct ports belonging to two distinct components) with the use of n-ary connectors.

In the work developed during the PhD of Nicolas Desnos, components enhanced with ports (that represent dependencies between several provided and required interfaces of a component) help identify the smallest interface set to connect to have an assembly that is correctly connected to handle some desired functionality. Ports thus embody the set of interfaces involved in the collaboration two components establish in order to implement a given functionality. Ports are a peer-to-peer means to connect components (one port is to be connected to a single compatible port). Connecting ports in one step

(atomically), instead of connecting each pair of interfaces in turn, decreases complexity.

Port compatibility is an answer to research question q4: what are the interfaces of a given component that have to be connected? After having identified the wished functionality (we call the functional objective) in an interface, the ports that contain this interface have to be connected.

In Figure 2.10(a), ports are connected through a valid connection. Figure 2.10(b) illustrates an impossible connection as it is not peer-to-peer. A solution to this issue might resemble Figure 2.10(c) if the model had a concept to signify that connecting one port of the :ATM component implies connecting the :ATM's other port. This is made possible using the new notion of composite port. Standard ports are now said to be **primitive ports**.

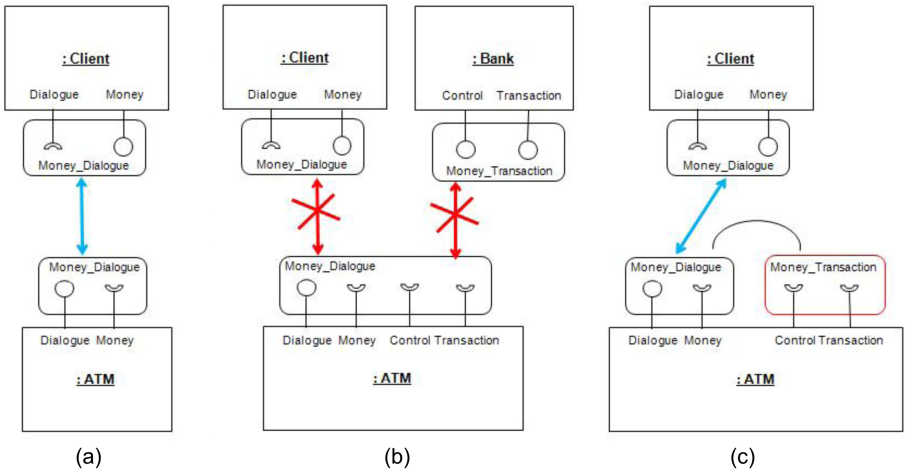


Figure 2.10: Primitive port peer-to-peer atomic connection

Composite ports offer a new structure to model complex collaborations. Composite ports combine a set of ports, primitive or composite, that all must be connected simultaneously but need not be connected to a unique port / component (multi-peer non-atomic connections). Composite ports can be seen as a means to express dependencies between ports. A connected composite port is shown on Figure 2.11.

Primitive and composite ports form a full toolset to express dependencies between interfaces. Combining primitive and composite ports makes it possible to express complex dependency schemes between interfaces. Ports provide information on correct interface connection being given the functional objectives of the software (research question q4). If all ports of all its components are either not connected or correctly connected, an assembly is in turn correctly connected: the property that stands for ports can be composed to calculate the same property for component assemblies. This answers research question q5: when is a component assembly correctly (fully) connected?

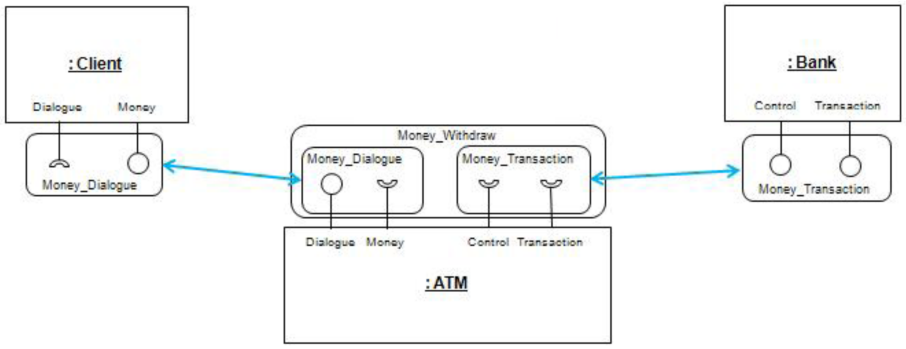


Figure 2.11: Composite port multi-peer non-atomic connection

As compared to models that connect all a component's interfaces or to models that define *a priori* optional interfaces, ports are more expressive and make it possible to define more precise connection schemes for components in an architecture. This precision both eases reuse and increases reusability. As compared to calculating the compatibility of the dynamics of components (*e.g.*, expressed as protocols), port compatibility and port connection computes much easier. This benefit is obtained at the expense of providing extra documentation for components: the structuring of their external view with ports. Compared to providing collaboration protocols, ports are easier to model. Ports do not substitute to defining collaboration protocols: both notions might be complementary. Furthermore, ports might probably be calculated from protocols, where dependencies (which ports model) are expressed altogether with time sequencing (which ports do not model). Calculating ports once and for all and saving the result of the calculus in the component repository would further decrease the computing cost of calculating whether an assembly is correctly connected or not.

The property of correctly connected component assemblies has been described formally using a formal set-theory inspired *ad hoc* notation [43, 42].

2.4.3 Port-guided automatic components assembling

Starting from identified functionalities (the requirements) components can be automatically connected to form coherent assemblies, meaning assemblies where all necessary ports are coherently connected.

The functionalities identified as objectives are a rudimental way to express the wished architecture (research question q2: how is the wished architecture going to be expressed?). Starting from this data, components are searched for that provide these functionalities. The selection of these components create dependencies that have to be satisfied (their required interfaces) which in turn leads to components that provide these functionalities and so on. Dependencies are all satisfied in turn until none remain

unsatisfied which means that the assembly is correctly connected. Instead of taking into account all of the required interfaces of a component to determine the inferred dependencies, only those embedded in the same port as already connected interfaces are considered.

By this means, correctly connected component assemblies can be generated by exploring all connections possibilities with all the components from the component repository. All correct assemblies can theoretically be generated using the port connection property and then discriminated amongst by using the more costly components dynamics comparison [92]. At this point, architects could also be asked to select assemblies that they consider pertinent to solve their problem.

Despite the simplification of the connection scheme, the automatic connection exploration is a complex process. Combinatorial explosion is tamed in practical cases (in which the size of architectures never exceed a reasonable number of components) by the use of problem-specific heuristics [42] (*e.g.*, search for the smallest, in terms of number of components, possible solutions).

Experiments with artificially generated component external descriptions with ports showed that the proposed property and assembly exploration algorithm makes it possible to give a positive answer to research question q6: can components be assembled automatically? To the best of my knowledge, such a search-based approach for components assembling is quite a novel contribution even if a close subject is much more commonly explored in the paradigm of web services where composition is automated using planning techniques [31, 72, 41].

2.5 Software architecture evolution

Handling evolution is a core component-based software engineering activity. It is considered to concentrate costs and to be the longest activity of the engineering process as it spans the whole life-cycle of the software. It is also multi-facetted. This section presents the several viewpoints of software architecture evolution I have explored.

2.5.1 Issues of software architecture evolution

Architecture evolution is a multi-facetted topic [25, 28].

It can be studied from a theoretical point of view, trying to identify what is the object of change and what is the motivation of the change or a pragmatic point of view, trying to provide concrete tools to implement evolution [71]. Some works [30, 78] define a taxonomy of changes, thus decomposing evolution in more focused fields or proposing to treat changes according to a priority order. Some works concentrate on managing the history of the changing software by maintaining its versions [35].

Unmanaged changes may cause architecture degradation. Architecture drift and erosion [91] are well known manifestations of such degradations.

We set as a manifesto that evolutions cannot all be anticipated and enumerated extensively. This would be equivalent to knowing in advance the future of the software. Such predictable changes would considerably ease the management of architectures but are not *per se* changes but rather functioning modes of the software. Real changes cannot be anticipated. This is an answer to research question q7 . This is also the reason of the existence of such mechanisms to control the unanticipated evolution of the software architecture and of the software itself this section is going to present.

2.5.2 Version propagation along relations

Model versions are recognized to be a medium for tracing evolution and maintaining data coherent.

Inspired from well established source code versioning systems, model versions were first widely studied in the context of databases in late 1980's and early 1990's. They applied at both data (which can be compared to instances in the object-oriented paradigm) and database schema (which can be compared to the class level model in the object-oriented paradigm) [86, 96]. Model version were also particularly used in computer-aided design applications [33, 108] in a wide range of domains such as VLSI, telecommunication networks, mechanics, architecture, software engineering, hypertext or multimedia creation. These applications were characterized by design steps made up of parallel proposals representing various points of view on the system (collaborative work) which needed to be validated, compared, sometimes reconciled (merged). Moreover, the data structures dealt with by these applications are complex and involve several distinct relations with their domain-specific semantics.

In this context, the aim of my PhD thesis was to define a version model which tackled the complex nature of the modelled entities, using the concepts defined in object systems.

The proposed version model, which is intended for application designers, proposes a parameterizable means to manage data evolution. This approach differs from the one usually chosen by version models, which provide fixed version management capabilities. The proposed complex entity version management model consequently divides into two user-levels: microscopic and macroscopic. Depending on their needs and on those of the targeted application, designers can choose the level they consider to suit them best. The microscopic level is intended for expert users who have specific version management needs, while the macroscopic level needs less parameterization.

Moreover, the proposed model allows any type of complex entity to be versioned, as it has not been defined for a particular category of dependency relations. It is parameter-

izable: existential dependencies are dealt with by an operation propagation mechanism described by the means of propagation rules and strategies. Designers can either reuse predefined rules and / or strategies or create new ones in order to correspond to a given objective.

Propagation strategies group active propagation rules which obey to the Event Condition Action (ECA) structure [47]. They are associated to relations. They define whether versioning one end of the relation implies versioning the other end. This version creation propagation mechanism is parameterizable using predefined or specific strategies or rules. In enables to maintain graphs of related model entities coherent as related to their versioning, as all versions of all entities might not be compatible with one another.

The proposed version creation propagation model is a partially affirmative answer to research question q9: can evolution be automated?

2.5.3 Many-to-one component substitution

When a component in an architecture misses or fails, there is a need to replace it by some equivalent. Most research work that do so search for a single substitute to provide the services of the removed component.

The port-enhanced components, presented in Section 2.4, are a means to replace a missing component (one) and its now unused dependencies by a component assembly (many) that fills the functional gap. This **many-to-one component substitution** mechanism uses the same technique as the one presented for composing architectures from components in a repository and is one of the results of the PhD of Nicolas Desnos [43, 42].

The replacement of the missing component can then be decomposed into two steps:

- Remove the defective component and its dependencies, called the dead components. When the component assembly without the defective component is considered as a graph, all connected components of the graph (its isolated subgraphs) that do not contribute to realizing a functionality listed in the architecture's objectives is composed of dead components (which used to be here only to fulfil the dependencies created by the defective component). These components can be removed to "clean up" the architecture and make it more simpler to replace the missing component by avoiding extra useless dependencies.

- Consider the incomplete component assembly as an intermediate result of our iterative building algorithm as presented in Section 2.4 and therefore run the building algorithm based on port-enhanced components on this incomplete assembly to re-build a complete assembly.

Figure 2.12 illustrates such a substitution on a simple evolution scenario. For simplicity's sake, instead of representing components as they usually are with all their connected interfaces, here, architectures are graphs, the nodes of which are components (shown as circles) and the edges of which are inter component connections (a single edge represents all dependencies).

When the (defective) `MemberBank` component is removed, the graph is analyzed and its connected components separated in two groups: the connected components that contribute to a functional objective (live components) and the connected components that were dedicated only to fulfilling the dependencies of the removed component (dead components). The dead components are removed and the assembly completed by adding the `Independent Bank` and `Bank IS` new components.

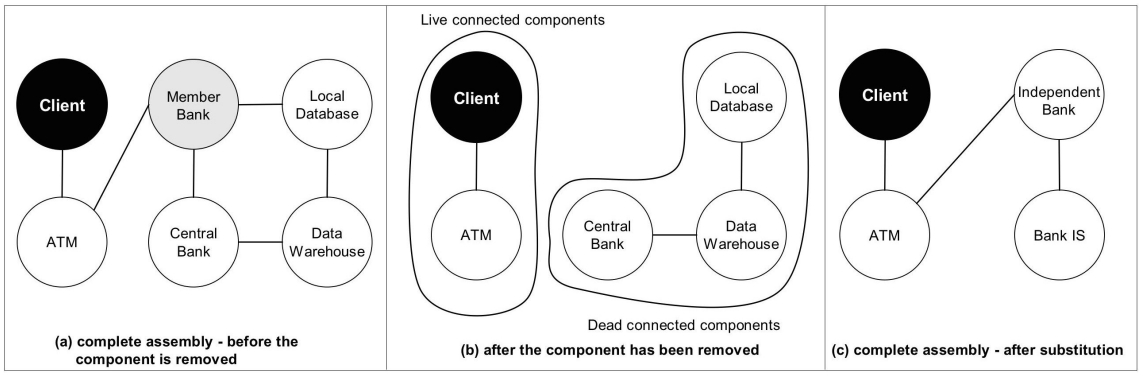


Figure 2.12: A many-to-one component substitution scenario [42]

If components are found in the repository that can together replace the missing component, the result is a restored architecture where one component has been changed for a component assembly of many smaller-grained ones. Such a process increases the chances to find a replacement component as it does not only search for complete coarse-grained components that implements at once all the missing functionalities but also for smaller grained components that will altogether be able to play this role. As the building algorithm searches for the smallest possible solutions (in terms of number of components), these two strategies are complementary: if a single component is not found, the smallest possible assembly will be searched for. Of course, the selected assembly can be stored in the repository as an already built solution for future (re)use.

The proposed many-to-one component substitution mechanism is an affirmative answer to research question q9: can evolution be automated?

This many-to-one component substitution mechanism can also be seen as an **automatic composition** mechanism that assembles smaller-grained components into a coarser-grained one. This coarser-grained component fulfils the objectives defined by

what previously was the missing component, here considered as an abstract component representing the composition objectives (the external description of the wished resulting component). Such an automatic composition mechanism would be useful to complete another step of the CBSE activities (*i.e.*, the `compose` activity of Figure 1.1) and should be integrated in a tooling that covers the whole software application development by component reuse process (see Section 3.3.4).

2.5.4 Automatic calculus of architecture evolution plans

Changes must be treated as first class entities. During the PhD of Huaxi (Yulin) Zhang an evolution management model that suits the three-level architectural model of Dedal [119, 118, 120] was proposed.

Changes are handled when initiated at any abstraction levels: specification, implementation and deployment. Their effects are applied locally and propagated to the neighbouring levels so as to maintain all descriptions coherent with one another. If changes are considered to disturb the equilibrium of the architecture descriptions, the propagation mechanism can be seen as a means to reach another point of equilibrium, that is another state where architectural descriptions are coherent. If the architecture specification is to be affected by the change, the change management system can either forbid the change (in a computer-empowered mode) or create a new coherent version of the whole architecture with modified specifications (in an architect-empowered mode).

During the PhD of Huaxi (Yulin) Zhang, evolution was managed in an *ad hoc* manner. Thanks to the formal ground of his work, the PhD of Aderrahman Mokni brings these ideas further. Indeed, a more disciplined and formal approach of evolution [81, 83, 82, 85] is proposed.

First, basic architecture change operations (*i.e.*, addition, deletion and modification of components and connections) are described formally at each of the architecture description levels. Then, evolution rules that react to changes by restoring the coherence of the architecture are formally defined. Finally, the B solver is used as a planning tool as the exploration of the rules to reach the objective of all description being consistent is equivalent to the compilation of a sequence of change operations. This sequence constitutes an **evolution plan** that is automatically searched for among all possible change operation sequences and restores the consistency of the architecture.

To overcome the inherent complexity of the search in practical cases, the generic B solver has been adapted using problem-specific heuristics. After being proposed for validation to the architect as a roadmap that describes the necessary changes consequent to an initial perturbation, the evolution plan can be transformed into an executable sequence of operations that are effectively executed on the architecture.

The proposed automatic calculus of architecture evolution plans is an affirmative

answer to research question q9: can evolution be automated?

Chapter 3

Conclusion and research project outlook

This conclusive chapter is going to present a synthesis of the contributions I have talked about in this dissertation. It also says a word on the work of PhD students and postdocs I did not include in this dissertation. The five research articles appended to this document are then introduced. To finish, some perspectives to this research are presented.

3.1 Synthesis

This dissertation showed how questions around the process of engineering and maintaining component-based applications have been the central theme of my research work so far.

During my PhD, I have worked on **evolution**, tracing the states of systems using a versioning mechanism that applied on models of both class and instance levels. The main contribution of this work is a co-evolution mechanism that could be parameterized in various ways using strategies and rules attached to relations that define how versioning had to propagate or not from a (changed) class or instance to its related ones that might be affected by the change or not. One of the relations studied explicitly is **composition**.

With Nicolas Desnos, I explored **composability**, defining the semantics of primitive and composite port connections to be able to automatically assemble components from a repository while connecting the smallest possible number of each component interfaces. I also explored means to automate **evolution** management when using the automatic assembly algorithm and ports semantics to repair an architecture configuration or an assembly in which a component misses (because it is unavailable or defective).

With Huaxi (Yulin) Zhang, I explored **reusabiliity**, defining a (third) system requirement level in our new Dedal ADL that serves as a perennial description of the architect's intention. This intention is exploited at design time to choose the component implementations that best match the wished ones and compose the architecture's configuration.

It also serves as a guide to the **evolution** process to determine if changes that occurred in the assembly or in the configuration result in a software that still matches the system requirements expressed in the architecture specification or if they imply either to forbid such changes or to check with the architects if he / she wants to modify the specification accordingly. A small Domain Specific Language (DSL) was also proposed to describe changes that both gathers information in a single place and serves for traceability.

With Abderrahman Mokni, I further explored these topics. Writing a formal semantics for the Dedal ADL and formalizing evolution rules gave us access to new tools for the management of component-based development. Indeed, the B formal language comes with automatic solvers and semi automatic provers that made it possible to check the well-formedness of (each of the three levels of) architecture descriptions and the validity of the semantics of inter-level relations thus acting as a complementary tool to support modelling and composition. The B formal language and tools were also used as a planner to repair an architecture configuration or an assembly after it being affected by changes. This newly tooled approach makes it more concrete to **reuse** components and **evolve** component-based software as it provides more automated help to architects in the core of their eclipse-based integrated development environment (IDE).

With Gabriela Arévalo, I explored automatic indexing of components according to their substitutability relation. This type-based indexing is a key mechanism of the efficient organization of component repositories and eases the querying of components that match some specification, can substitute to a component or can assemble to a component. It therefore is a cornerstone of all component-based development major tasks: indexing makes component **reuse** possible when repositories are too big, too numerous or too distributed for an expert to learn what components are available. Thanks to such indexing, components can be searched for in order to either be assembled to or replace some other component. It thus is a central mechanism for **composition** and **evolution**. The proposed type-based substitutability partial order is mainly based on syntactical information.

With Zeina Azmeh, I started to explore complementary (more semantical) means to index services by comparing meta-information extracted from their descriptors. This work used techniques inspired from natural language analysis and information retrieval techniques. These, combined with FCA, resulted in automatic service classification according to the proposed substitutability relation. Such classification was the pre-calculus of a partial order among services that was meant to enable to quickly search a replacement service in case one became unavailable. It is therefore clearly a contribution to service **reuse** by **composition** and service-based application **evolution**.

These contributions are all summed up in Figure 3.1.

In this dissertation, I did not present the work done with Rafat Al'Msiedeen on the retro-engineering of software product lines from the code of several software product

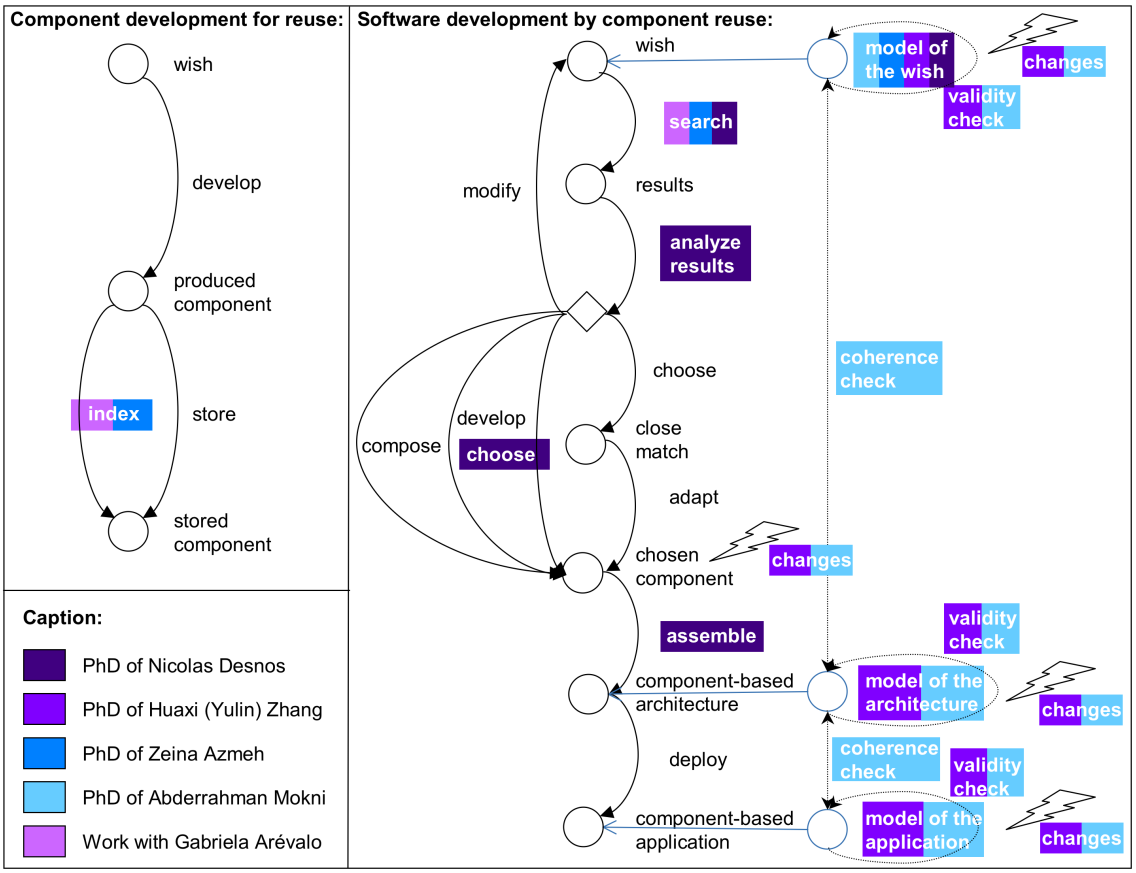


Figure 3.1: Synthesis of the presented contributions

variants [10, 11, 12, 7, 9, 8, 6, 5]. In this work, I explored deeper how information retrieval techniques (*e.g.*, Latent Semantic Indexing (LSI) [40]) could be used to mine information from textual data (here, software code) by assessing their similarity. This work is inspiring in two ways. First, it encourages me to further explore various means to classify components, using syntactical data as well as semantical data and furthermore, collecting extra data from usage or from users. This will be the basis for providing context-aware typing relations as described in Section 3.3.1. Second, it encourages me to further exploit the feature models designed to express commonalities and variabilities in software product lines in order to better describe variability in architectures. Indeed, having their system requirements described as a set of possibilities represented not solely by the types of the specified roles (and the component class to role matching relation) but also by a model that explicitly represents more complex variations in the admissible specifications would expand the possibilities of automatically building or evolving architectures as foreseen in Section 3.3.2.

In this dissertation, I did not present the work done with Guillaume Grondin either. During his post doctorate on the Hydroguard flood surveillance project, we worked on AROLD, a Domain Specific Language (DSL) to declaratively describe missions that had

to be achieved by the software system [56]. This language was adapted to describing several alternative missions that correspond to alternative objectives depending on the context. For example, the system had to monitor parameters measured by sensors and collect their values over time but when a parameter had an abnormal value, meaning something was going wrong on the field, the system had to automatically switch to a crisis mode in which its missions where different, directed to alert the authorities. The missions were automatically deployed based on existing implementations and their deployment optimized on the existing calculation units using a greedy approximation algorithm and problem-specific heuristics. This work was a first step into providing high level knowledge on the system behavior in order for it to autonomically adapt to changes in its environment. Such an idea could be exploited further with a feature-based vision of the specifications of architectures (see Section 3.3.2).

During their PhDs Fady Hamoui and Matthieu Faure both studied how component-based software engineering techniques could apply in pervasive environments to have control software self-adapt to its context. I chose not to include the presentation of their work in this dissertation even if I strongly believe pervasive environments are a choice application domain for my research. Indeed, in pervasive environments there are distributed devices that each can carry its own local component repository. There are also sensors and actuators that make it possible for the software to have inputs from its context and to concretely act on it. Mobility of users carrying devices and mobility of sensors and actuators as they come with equipment that can be added to or removed from the environment easily make applications that are developed for these environments in demand for runtime self-* mechanisms that strongly resemble those studied and described above. At last, Home Automation Applications are examples of applications that execute in such pervasive environment and they are very concerning as they can participate at increasing our comfort and security at home, better managing our energy consumption so as to preserve the environment or providing services tailored for the more fragile among us (elderly or disabled people). In this context, with Fady Hamoui, I explored how home monitoring applications could be described in a high level manner through rule-based scenarios. These scenarios are deployed and executed thanks to an agent-based system in which the software part of agents is self-assembled from generated components some of which control a sensor or an actuator and another of which is responsible for coordinating the execution of the scenario [59, 61, 60]. With Matthieu Faure [50, 52, 51], I proposed a DSL to declaratively describe scenarios. One of its characteristics is that it inherently adapts to its context with, for example, the *all* and *any* quantifiers that make it possible to describe actions such as *turn any of the dining room lamps on* or *shut all shutters*. This DSL is furthermore tooled with a composition engine that composes the software that is going to be executed and a step by step execution model that takes device (thus service) mobility into account in order to define scenarios that span over several places. Scenario sharing is also possible. As an application field, home automation still is a perspective for my research (see 3.3.4).

In this dissertation, I did not present the results of Frédéric Souchon's PhD either.

The objective of this work was to propose a fault tolerance mechanism in the form of an exception handling system adapted to distributed, asynchronous software such as agent systems or message-driven component-based applications [101, 103, 100, 102]. This work nonetheless is an incentive to fully integrate exceptions as a key information on functionality that impacts their substitutability (see Section 3.3.1).

3.2 Choice of appended articles

This synthesis of my contributions is accompanied by a selection of five research articles. They have been chosen so as to cover this overview. Most of them are long (journal) papers. As such, they provide deeper details on most aspects of this overview. They are appended in the chronological order of their publication:

1 – Christelle Urtado and Chabane Oussalah. Complex entity versioning at two granularity levels. *Information Systems*, 23(2/3):197–216, 1998. IF (1998) 1,547. ERA rank A*.

It is one of the achievements of my doctoral thesis. It sums up the vision I had on versioning as a mean to trace (co-)evolution. Indeed, it defines a versioning model that parameterizes using versioning strategies and rules how relations between concepts impact their being versioned together or not. See Appendix A.

2 – Nicolas Desnos, Marianne Huchard, Guy Tremblay, Christelle Urtado, and Sylvain Vauttier. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution: Research and Practice, Special issue on Search-Based Software Engineering*, 20(5):321–344, September / October 2008. IF (2008) 0,971. ERA rank B.

Published during the PhD of Nicolas Desnos, this journal article shows how primitive and composite ports, together with a companion algorithm to automatically assemble components can be used as a mean to drive evolutions that repair an assembly after a component failure. See Appendix B.

3 – Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Formal concept analysis-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, 38(4):427–453, May 2009. IF (2009) 0,611. ERA rank C.

This journal article reflects the results on component classification and component repository indexing that was developed with Gabriela Arévalo. See Appendix C.

4 – Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In Muhammad Ali Babar

and Ian Gorton, editors, *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, volume 6285 of LNCS, pages 295–310, Copenhagen, Denmark, August 2010. Springer. AR 25,3%. ERA rank A.

This conference paper was published during the PhD of Huaxi (Yulin) Zhang and introduces the Dedal ADL and its three architecture representation levels tailored for component reuse. See Appendix D.

5 – Abderrahman Mokni, Christelle Urtado, Sylvain Vauttier, Marianne Huchard, and Huaxi (Yulin) Zhang. A formal approach for managing component-based architecture evolution. *Science of Computer Programming, special issue of the 11th international symposium on Formal Aspects of Component Software*, 127:24–49, October 2016. IF (2014) 0,715. ERA rank A.

This journal article is one of the achievements of the PhD of Abderrahman Mokni. It shows that a B formalization of the Dedal ADL and formalized evolution rules make it possible to have evolution plans calculated automatically in order to restore an assembly after it being impaired by some change. See Appendix E.

3.3 Research project

My perspectives for this research are numerous but all converge towards the objective of proving the interest of setting a concrete software engineering based on components.

3.3.1 Context-aware and usage-aware component substitution relations

Substitution and compatibility relations are central to component search, connection and replacement. Several definitions are possible for the substitution relation: from strict equality, to strict typing in the sense compilers do, to less strict type-based proposals that admit small adaptations (parameter duplication or re-orderings), to semantic typing relations inspired from information theory. All the possible information natures identified in Section 2.2 might lead to several pertinent and yet unexplored combinations thus defining interesting typing relations. As an example, it is not easy to take functionality raised exception types into account as the semantics of sub-typing a required functionality that can signal an exception type is not obvious.

In an adaptable system, all these relations could all be available and the best that suits some component search context be adopted automatically. We could distinguish two situations:

- **searching in the large**. There are contexts where there are a lot of components. Component search has to deal with the heterogeneity of the component sources

and their being numerous. It might be useful to have a strict substitutability relation that will strongly discriminate components and identify components that strictly match requests. The fact that components come from various providers, might be addressed by considering semantics-based relations to mitigate different vocabularies and thus constitute a first filter before applying type-based selection methods. Ontologies might be used here, for example [97, 57]. Discriminating among components might also mean strengthening the search constraints, asking for higher quality of service or availability and comparing costs to choose the lowest.

- **searching in the small**. On the opposite, there are contexts where components are scarce. In order to still find something (and not redevelop unless it is really needed), a less strong substitutability relation that will find extra close matches might be used. Resulting components might not be exactly what would initially have been searched for but they might become suitable for reuse after small adaptations. These adaptations might be automated in some simple cases or manual. Adaptations should be preferred to development from scratch as long as they are less costly [79].

Such a system would imply to have metrics to qualify the search context and make decisions accordingly. Metrics could be calculated on metadata provided by developers when they store components in the repository or obtained automatically (*e.g.*, by querying the repository with some requests and generalizing from them).

Ultimately users might be involved in a sort of collaborative social network in which experiences with component reuse might be shared in several ways and constitute an aggregated rating that creates a feedback loop and parameterizes future searches. For example, tracking component usage (*e.g.*, usage frequency, number of tries for a same request) or having users rank or evaluate the components they use might be user-centered information used to increase trust in component choices. There might also be an automatic recommender system that scans your usages and determines your profile in order to proactively propose new components that might suit your needs.

Such ideas have been rapidly outlined in one of my prospective research paper [15].

3.3.2 Feature-based architecture requirements

System requirements are the first entry of the software development by component reuse process. In the proposed system, they are expressed by the means of abstract architecture specifications. These are a flexible expression of what is needed as compared to many models. Indeed, in most of them, solely configurations exist that do not permit to make changes to them in the range of broader specifications (except if enumerating extensively all valid configurations which is a "closed" means to define possible evolutions). Abstract architecture specifications define what the architecture should look like intently as roles and the type-to-role matching relation state what is the minimum set

of functionalities that should be implemented but do not make the set of possibilities an extensively enumerated one.

The expression of system requirements might yet be more flexible if it could be based on the definition of mandatory or optional roles (and maybe even include constraints between those roles to express co-occurence or mutual exclusion).

Using features models [66] as a complement of abstract architecture specifications [21, 65] might be an interesting means to define more complex (and larger) sets of possible configurations that meet the requirements while still defining them intently (as opposed to extensively enumerating all possibilities).

Feature models are used to specify commonalities and variability when defining software product lines. In the case of feature-based system requirements, they might define various types of configurations, that each could be considered as a specific product in the architecture product line.

3.3.3 Seeding the reuse process by reverse-engineering components and architectures

In order to seed and promote such a component reuse process, there is a need for a sufficiently rich component offer. The idea is to have components in an adequate number to start constitute a panel of reuse possibilities. There is also a need for material to analyze in order to experiment our ideas on concrete cases and validate them. As such repositories still do not exist, there is a need to seed the process of concretely adopting a component-based engineering development process. This can be done by mining existing projects. Three sorts of data could be searched for and exploited:

- **components or architecture fragments**. The idea is also to have the components be adequately documented, with their ports for example, so as to be able to adequately connect them. To identify components from existing non component projects, it is possible to reverse engineer existing open-source projects. It is easier if these are written using an object-oriented language.

  A rough approximation would be to consider a class as a component and analyze its methods and dependencies to respectively determine its provided or required interfaces. If there are no interfaces in the code (either they have not been programmed or the concept does not exist in the chosen programming language), there will be a need to cluster provisions and requirements in order to define these interfaces.

  If assimilating a class to a component looks unsatisfactory, classes will have to be clustered from code into components [67] using some heuristics and metrics (such as cohesion and coupling [46]). An alternative way does not need that the code be

available and clusters object-oriented APIs into components from their usages [98].

If hierarchical, these approach might detect (composite) components that themselves are architecture fragments. Storing both detail levels – the inner components and the architecture fragment that describes a coarser-grained component – is interesting in that it provides components of various granularities and increases both the probability of finding a suitable component that does not provide too much extra functionalities (in order not to over-constrain the remaining of the search) and the probability of finding sufficiently coarse components to suit our needs (not to have to take the risk of rebuilding them from smaller ones).

- **architectures**. The idea is also to have adequately documented architectures so as to be able to test our proposals against. The three levels of architecture descriptions have to be mined, especially their specifications. Architecture specification will be a guide for architecture evolution or will serve to *a posteriori* validate past evolutions that occurred during the life of the studied application.

  In a first approach, classes can be assimilated to components if their dependencies can be discovered and exposed. More precise approaches can also apply as described above. Application description in the form of declarative scripts are close to configuration descriptions. Such scripts (XML assembly descriptors in J2EE, Spring scripts and, maybe, build scripts) could be used to automatically generate configurations by reverse engineering real open-source applications. This is a lead to explore for mining configurations but other approaches could be studied as classified in Stéphane Ducasse *et al.*'s literature review [49].

  The assembly level description and instance constraints can be searched for in the code: even if it might not be easy to explore all possible instantiations in theory, I hope, in practice, instantiation patterns are limited.

  The remaining problem, that is specific to the presented approach, is the need for an architecture specification description. For this level, there might not be any documentation to mine the information from. In a first (restrictive) approximation, specifications can be the exact definition of the configurations found. It might be more interesting to mine them from several resembling configurations. If such various configurations exist, classification techniques such as FCA could be used for this purpose. Abstract concepts that supersed a group of concept representing existing configurations might be good descriptions of specifications that undergo this set of configurations.

- **component and architecture histories**. In order to be able to test our ideas against real evolution scenarios, there is also a need to mine version histories of components and architectures. This will make it possible to replay realistic changes and confront our tools to real evolutions. This will also serve to encourage their

use in a forward component-based software engineering process.

A key issue will be to improve version management of components and architecture descriptions in our proposal. This is necessary in order to better manage inter-version (in)coherence or propose a framework where several versions of a given component might co-exist in a single version of an architecture.

Having adequately documented components and architectures as well as their histories would be the first step towards developing wider range experiments of automatic component assembly (and comparing the results to real architectures), architecture coherence verification and architecture evolution (replay of real mined evolution scenarios). A well supplied repository would also become an argument to promote component and architecture fragment reuse.

Several sources might be used to reverse engineer components and architectures such as: long-lived object-oriented open-source projects (*e.g.*, ArgoUML[1]), projects that already include components (*e.g.*, Fractal[2] or FraSCAti[3] projects), Spring open-source projects, etc.

3.3.4 Tooling and applications

In order to better promote the ideas presented in this manuscript, there is a need to pursue the effort towards having a single demonstrator that embeds all the ideas developed here.

The different steps and activities of software application development by component reuse have to be completed so as to compose a continuous process (see Figure 3.1 for activities that have not been yet tackled). Conversely, when several proposals for a given activity coexist in my proposed works, they have to be merged or one must be chosen (see also Figure 3.1 for activities where several proposals exist).

As an implementation platform, the choice to tool the eclipse IDE has become our option after having worked around the Julia implementation of the Fractal component model and ADL (during the 2005-2010 frame period). Eclipse is both a well known (*de facto* standard) IDE and an extensible application, tooled with numerous plugins that help constitute a full tool suite without the needs of reinventing what already exists.

We can, for example, rely on frameworks and tools we have already used:

- the Eclipse Modeling Framework (EMF) meta-model[4] to support the concepts of the proposed component model (*e.g.*, primitive and composite ports),

[1] `http://argouml.tigris.org`.
[2] `http://fractal.ow2.org`.
[3] The FraSCAti component framework implements the Service Component Architecture (SCA) specification. `http://wiki.ow2.org/frascati/`.
[4] `https://eclipse.org/modeling/emf/`.

- the Eclipse Sirius[5] platform to create original graphical modeling tools (DSLs with their own graphical concrete syntax) on top of EMF,
- the ProB API that provides access to B solver capabilities (animation and model-checking) and makes it possible to code a dedicated specific solver,
- the OSGi service platform[6] to manage component deployment and host their execution.

The prototype developed during the PhD of Abderrahman Mokni, called the Dedal-Studio tool suite, can be a starting point for this tooling integration.

The application domain of pervasive, cyber-physical systems, meaning systems that mix software and hardware devices or things (sensors, actuators, robots, etc.) is a context where software would better be self-* and context-sensitive. It is a very inspiring application field for the proposed software engineering techniques. Focusing on home automation for the elderly or disabled people as well as setting a software architecture that makes it possible to use connected things, as named by the Internet of Things (IoT) trend, to propose new services while preserving the privacy of the collected data is a real challenge.

[5] https://eclipse.org/sirius/.
[6] https://www.osgi.org/.

Bibliography

[1] Nour Aboud, Gabriela Arévalo, Olivier Bendavid, Jean-Rémy Falleri, Nicolas Haderer, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Building hierarchical component directories. Submitted to Journal of Object Technology. 34 pages, August 2014.

[2] 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 *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (WICSA / ECSA 2009)*, pages 21–30, Cambridge, UK, September 2009. IEEE.

[3] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[4] Jiri Adamek and František Plášil. Partial bindings of components - any harm? In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 632–639, Busan, Korea, November 2004. IEEE.

[5] Rafat AL-Msie'deen, Marianne Huchard, Abdelhak Djamel Seriai, Christelle Urtado, and Sylvain Vauttier. Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach. *International Journal of Software Engineering and Knowledge Engineering*, 24(10):1413–1438, December 2014.

[6] Rafat AL-Msie'deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, and Sylvain Vauttier. Reverse engineering feature models from software configurations using Formal Concept Analysis. In Karell Bertet and Sebastian Rudolph, editors, *Proceedings of the 11th international conference on Concept Lattices and their Applications (CLA 2014)*, volume 1252 of *CEUR Workshop Proceedings*, pages 95–106, Košice, Slovakia, October 2014.

[7] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In Chengcui Zhang, James Joshi, Elisa Bertino, and Bhavani Thuraisingham, editors, *Proceedings of the 14th IEEE international conference on Information Reuse and Integration (IRI 2013)*, pages 586–593, San Francisco, USA, August 2013. IEEE.

[8] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Documenting the mined feature implementations from the object-oriented source code of a collection of software product variants. In *Proceedings of the 26th international conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, pages 138–143, Vancouver, Canada, July 2014.

[9] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Ahmad Khlifat. Concept lattices: a representation space to structure software variability. In *Proceedings of the 5th International Conference on Information and Communication Systems (ICICS 2014)*, page 6, Irbid, Jordan, April 2014. IEEE.

[10] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. An approach to recover feature models from object-oriented source code. In *Actes de la Journée Lignes de Produits 2012*, pages 15–26, Lille, France, Novembre 2012.

[11] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature location in a collection of software product variants using formal concept analysis. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Reuse - Proceedings of the 13th International Conference on Software Reuse*, number 7925 in LNCS, pages 302–307, Pisa, Italy, June 2013. Springer.

[12] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing. In *Proceedings of the 25th international conference on Software Engineering and Knowledge Engineering (SEKE 2013)*, pages 244–249, Boston, USA, June 2013.

[13] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 187–197, Orlando, USA, May 2002. ACM.

[14] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[15] Gabriela Arévalo, Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Component and service farms. In Eric Cariou, Laurence Duchien, and Yves Ledru, editors, *Actes des deuxièmes journées nationales du GdR Génie de la Programmation et du Logiciel — Défis pour le Génie de la Programmation et du Logiciel*, pages 281–284, Pau, France, Mars 2010.

[16] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Precalculating component interface compatibility using FCA. In Jean Diatta, Peter Eklund, and Michel Liquière, editors, *Proceedings of the 5th international conference on Concept Lattices and their Applications (CLA 2007)*, pages 241–252. CEUR Workshop Proceedings Vol. 331, ISSN 1613-0073, Montpellier, France, October 2007.

[17] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services. In Y. Aït-Ameur, editor, *Actes de la 2ème Conférence francophone sur les Architectures Logicielles (CAL 2008)*, Revue des Nouvelles Technologies de l'Information (RNTI-L-2), ISSN 1764-1667, pages 123–138. Editions Cépaduès, Montréal, Canada, March 2008. Cet article a été accepté conjointement à LMO 2008. AR 53,6%.

[18] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services. In Mireille Blay-Fornarino, Yann-Gaël Guéheneuc, and Houari Sahraoui, editors, *Actes de la 14ème conférence francophone sur les Langages et Modèles à Objets 2008 (LMO 2008)*, Revue des Nouvelles Technologies de l'Information (RNTI-L-1), ISSN 1764-1667, page 39, Montréal, Canada, Mars 2008. Editions Cépaduès. Cet article a été accepté conjointement à CAL 2008 dans les actes de laquelle figure le texte complet. AR 40,5%.

[19] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services utilisant l'analyse formelle de concepts. In Yves Ledru and Marc Pantel, editors, *Actes des journées nationales du GdR Génie de la Programmation et du Logiciel*, pages 130–131, Toulouse, France, Janvier 2009. ENSEEIHT-IRIT.

[20] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Formal concept analysis-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, 38(4):427–453, May 2009.

[21] Trinidad Martín Arroyo, Pablo, Ruiz Cortés, Antonio, Peña Siles, Joaquín, and David F. Benavides Cuevas. Mapping feature models onto component models to build dynamic software product lines. In *Proceedings of the first Dynamic Software Product Line workshop @ the 11th Software Product Lines Conference*, pages 51–56, Kyoto, Japan, September 2007. IEEE.

[22] Zeina Azmeh, Fady Hamoui, Marianne Huchard, Nizar Messai, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Backing composite web services using Formal Concept Analysis. In Robert Jäschke and Petko Valtchev, editors, *Proceedings of the 9th International Conference on Formal Concept Analysis (ICFCA 2011)*, number 6628 in LNCS / LNAI, pages 26–41, Nicosia, Cyprus, May 2011. Springer.

[23] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. WSPAB: A tool for automatic classification & selection of web services using formal concept analysis. In *Proceedings of the 6th IEEE European Conference on Web Services (ECOWS 2008)*, pages 31–40, Dublin, Ireland, November 2008. IEEE.

[24] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Using concept lattices to support web service compositions with backup services. In *Proceedings of the 5th International Conference on Internet and Web Applications and Services (ICIW 2010)*, pages 363–368, Barcelona, Spain, May 2010. IEEE.

[25] Olivier Barais, Françoise Anne Le Meur, Laurence Duchien, and Julia Lawall. *Software Architecture Evolution*, pages 233–262. Springer, 2008.

[26] Marc Barbut and Bernard Monjardet. *Ordre et classification : algèbre et combinatoire*. Hachette, 1970.

[27] Boualem Benatallah, Mohand-Said Hacid, Alain Leger, Christophe Rey, and Farouk Toumani. On automating web services discovery. *The VLDB Journal*, 14(1):84–96, 2005.

[28] Hongyu Pei Breivold, Ivica Crnković, and Magnus Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16 – 40, 2012.

[29] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP 2002)*, Malaga, Spain, 2002.

[30] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September 2005.

[31] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO 2005)*, pages 1069–1075, Washington DC, USA, 2005. ACM.

[32] Michel R. V. Chaudron and Ivica Crnković. *Software Engineering; Principles and Practice*, chapter Component-based Software Engineering, pages 605–628. John Wiley & Sons, 2008.

[33] Hong-Tai Chou and Won Kim. A unifying framework for version control in a cad environment. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *Proceedings of the 12th International Conference on Very*

Large Data Bases (VLDB 1986), pages 336–344, Kyoto, Japan, August 1986. Morgan Kaufmann.

[34] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.

[35] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[36] Marco Crasso, Alejandro Zunino, and Marcelo Campo. AWSC: An approach to web service classification based on machine learning techniques. *Inteligencia Artificial*, 12(37):25–36, March 2008.

[37] Ivica Crnković, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, page 44, Papeete, French Polynesia, October 2006. IEEE.

[38] Ivica Crnković, Judith Stafford, and Clemens Szyperski. Software components beyond programming: From routines to services. *IEEE Software*, 28(3):22–26, May / June 2011.

[39] Luca De Alfaro and Thomas A Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[40] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, September 1990.

[41] Shuiguang Deng, Bin Wu, Jianwei Yin, and Zhaohui Wu. Efficient planning for top-k web service composition. *Knowledge and Information Systems*, 36(3):579–605, 2013.

[42] Nicolas Desnos, Marianne Huchard, Guy Tremblay, Christelle Urtado, and Sylvain Vauttier. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution: Research and Practice, Special issue on Search-Based Software Engineering*, 20(5):321–344, September / October 2008.

[43] Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay. Automated and unanticipated flexible component substitution. In H. W. Schmidt et al., editors, *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)*, volume 4608 of *LNCS*, pages 33–48, Medford, USA, July 2007. Springer.

[44] Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. Assistance à l'architecte pour la construction d'architectures à base de composants. In Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors, *Actes de la 12ème conférence francophone sur les Langages et Modèles à Objets (LMO 2006)*, pages 37–52, Nîmes, France, March 2006. Hermès.

[45] Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard. Automating the building of software component architectures. In Volker Gruhn and Flavio Oquendo, editors, *Proceedings of the 3rd European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications*, volume 4344 of *LNCS*, pages 228–235, Nantes, France, September 2006. Springer.

[46] Harpal Dhama. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, 29(1):65–74, April 1995.

[47] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto: A rulebase of adbms features. In Timos Sellis, editor, *Proceedings of the 2nd Workshop on Rules in Databases (RIDS 1995)*, volume 985 of *LNCS*, pages 3–20, Athens, Greece, September 1995. Springer.

[48] Xin Dong, Alon Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for web services. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the 30th international conference on Very Large Data Bases (VLDB 2004)*, pages 372–383, Toronto, Canada, August / September 2004. Morgan Kaufmann.

[49] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, April 2009.

[50] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Towards scenario creation by service composition in ubiquitous environments. In Stéphane Ducasse, Laurence Duchien, and Lionel Seinturier, editors, *Proceedings of the 9th edition of the BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, pages 145–155, Lille, France, December 2010.

[51] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. A service component framework for multi-user scenario management in ubiquitous environments. In *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA 2011)*, pages 155–160, Barcelona, Spain, October 2011.

[52] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. User-defined scenarios in ubiquitous environments: creation, execution control and sharing. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, pages 302–307, Miami, USA, July 2011.

[53] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1st edition, 1997.

[54] David Garlan, Robert T. Monroe, and David Wile. Acme: architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–68. Cambridge University Press, 2000.

[55] Dimitra Giannakopoulou, Corina S. Päsäreanu, and Howard Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE international conference on Automated Software Engineering (ASE 2002)*, pages 3–12, Edinburgh, UK, September 2002.

[56] Guillaume Grondin, Matthieu Faure, Christelle Urtado, and Sylvain Vauttier. Mission-oriented autonomic configuration of pervasive systems. In *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA 2012)*, pages 685–690, Lisbon, Portugal, November 2012.

[57] Suresh Chand Gupta and Ashok Kumar. Reusable Software Component Retrieval System. *International Journal of Application or Innovation in Engineering and Management*, 2(1):187–194, January 2013.

[58] Florian Hacklinger. JavaA - taking components into java. In *Proceedings of the ISCA 13th international conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)*, pages 163–168, Nice, France, July 2004.

[59] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Specification of a component-based domotic system to support user-defined scenarios. In *Proceedings of 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, pages 597–602, Boston, USA, July 2009.

[60] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. SAASHA: a Self-Adaptable Agent System for Home Automation. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, pages 227–230, Lille, France, September 2010. IEEE.

[61] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Un système d'agents à base de composants pour les environnements domotiques. In Eric Cariou and Jean-Claude Royer, editors, *Actes de la 16ème conférence francophone sur les Langages et Modèles à Objets (LMO 2010)*, pages 35–49, Pau, France, Mars 2010.

[62] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[63] George T. Heineman and William T. Councill. *Component-Based Software Engineering: putting the pieces together*. Addison-Wesley, 2001.

[64] Andreas Heß and Nicholas Kushmerick. Learning to attach semantic metadata to web services. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors,

Proceedings of the International Semantic Web Conference, volume 2870 of *LNCS*, pages 258–273. Springer, 2003.

[65] Mikoláš Janota and Goetz Botterweck. Formal approach to integrating feature and architecture models. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering: Proceedings of the 11th International Conference on Formal Aspects of Software Engineering (FASE 2008), Budapest, Hungary, March April 2008*, number 4961 in LNCS, pages 31–45. Springer, 2008.

[66] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA): Feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990.

[67] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny, and Allaoua Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Proceedings of the Joint 10th Working IEEE / IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA / ECSA 2012)*, pages 181–190, Helsinki, Finland, August 2012.

[68] Natallia Kokash. A comparison of web service interface similarity measures. In Loris Penserini, Pavlos Peppas, and Anna Perini, editors, *Proceeding of the 3rd Starting AI Researchers' Symposium (STAIRS 2006)*, volume 142 of *Frontiers in Artificial Intelligence and Applications*, pages 220–231, Riva del Garda, Italy, 2006. IOS Press.

[69] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[70] Meir M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process technology (EWSPT 1996)*, volume 1149 of *LNCS*, pages 108–124. Springer, 1996.

[71] Meir M. Lehman and Juan Carlos Fernandez-Ramil. Towards a theory of software evolution - and its practical impact. In *Proceedings of the International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 2–11, November 2000.

[72] W. Li, X. Dai, and H. Jiang. Web services composition based on weighted planning graph. In Lican Huang, Lean Yu, Maozhen Li, Junwei Cao, and Yuanan Liu, editors, *Proceedings of the first International Conference on Networking and Distributed Computing (ICNDC 2010)*, pages 89–93, Hangzhou, China, October 2010. IEEE.

[73] Wei Li and Sallie Henry. Object-oriented software object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993.

[74] Barbara H. Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Notices*, 23:17–34, January 1987.

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

[76] M. Douglas McIlroy. Mass-produced software components. In Peter Naur and Brian Randell, editors, *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, October 1968,*, pages 138–151, 1969.

[77] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[78] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software Evolution*, pages 1–11. Springer, 2008.

[79] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[80] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Modélisation et vérification formelles en B des architectures logicielles à trois niveaux. In Catherine Dubois, Nicole Levy, Marie-Agnès Peraldi-Frati, and Christelle Urtado, editors, *Actes de la troisième Conférence en IngénieriE du Logiciel (CIEL 2014)*, pages 71–77, Paris, France, Juin 2014.

[81] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Towards automating the coherence verification of multi-level architecture descriptions. In *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA 2014)*, pages 416–421, Nice, France, October 2014.

[82] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. An evolution management model for multi-level component-based software architectures. In *Proceedings of the 27th international conference on Software Engineering and Knowledge Engineering (SEKE 2015)*, pages 674–679, Pittsburgh, USA, July 2015.

[83] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Formal rules for reliable component-based architecture evolution. In Ivan Lanese and Eric Madelaine, editors, *11th international symposium on Formal Aspects of Component Software (FACS 2014), September 2014, Bertinoro, Italy, Revised Selected papers*, volume 8997 of *LNCS*, pages 127–142. Springer, 2015.

[84] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. A three-level formal model for software architecture evolution. In Davide Di Ruscio and Vadim Zaytsev, editors, *Post-proceedings of the 7th Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE*

2014), volume 1354 of *CEUR Workshop Proceedings*, pages 102–111, L'Aquila, Italy, July 2015.

[85] Abderrahman Mokni, Christelle Urtado, Sylvain Vauttier, Marianne Huchard, and Huaxi (Yulin) Zhang. A formal approach for managing component-based architecture evolution. *Science of Computer Programming, special issue of the 11th international symposium on Formal Aspects of Component Software*, 127:24–49, October 2016.

[86] Simon Monk and Ian Sommerville. Schema evolution in oodbs using class versioning. *SIGMOD Records*, 22(3):16–22, September 1993.

[87] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[88] OASIS. Web services business process execution language version 2.0, http://docs.oasis-open.org/wsbpel/2.0/os/wsbpel-v2.0-os.html, 2007.

[89] OMG. Unified modeling language, version 2.5, version 4.0 ,http://www.omg.org/spec/uml/2.5/pdf/, 2015.

[90] Chabane Oussalah and Christelle Urtado. Complex object versioning. In A. Olivé and J. A. Pastor, editors, *Proceedings of the 9th international Conference on Advanced Information Systems Engineering, (CAiSE 1997)*, volume 1250 of *LNCS*, pages 259–272, Barcelona, Spain, June 1997. Springer.

[91] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[92] František Plášil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

[93] Christian Platzer and Schahram Dustdar. A vector space search engine for web services. In *Proceedings of the third IEEE European Conference on Web Services (ECOWS 2005)*, pages 62–71, Växjö, Sweden, November 2005. IEEE.

[94] Ralf H. Reussner. Counter-constrained finite state machines: A new model for component protocols with resource-dependencies. In William I. Grosky and František Plášil, editors, *SOFSEM 2002: Theory and Practice of Informatics: 29th Conference on Current Trends in Theory and Practice of Informatics Milovy, Czech Republic, November 22–29, 2002 Proceedings*, volume 2540 of *LNCS*, pages 20–40. Springer, Berlin, Heidelberg, 2002.

[95] Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning about software architectures with contractually specified components. In Alejandra Cechich, Mario Piattini, and Antonio Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*, pages 287–325. Springer, 2003.

[96] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383 – 393, 1995.

[97] Nedhal A. Al Saiyd, Intisar A. Al Said, and Ahmed H. Al Takrori. Semantic-Based Retrieving Model of Reuse Software Component . *International Journal of Computer Science and Network Security*, 10(7):154–161, July 2010.

[98] Anas Shatnawi, Abdelhak-Djamel Seriai, Houari Sahraoui, and Zakarea Alshara. Reverse engineering reusable software components from object-oriented apis. *To appear in Journal of Systems and Software*, 2016.

[99] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[100] Frédéric Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. A proposition for exception handling in multi-agent systems. In *Proceedings of the 2nd international workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003)*, pages 136–143, Portland, USA, May 2003.

[101] Frédéric Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Improving exception handling in multi-agent systems. In Carlos José Pereira de Lucena, Alessandro F. Garcia, Alexander B. Romanovsky, Jaelson Castro, and Paulo S. C. Alencar, editors, *Software engineering for multi-agent systems II, Research issues and practical applications*, volume 2940 of *LNCS*, pages 167–188. Springer, February 2004. Selected, extended and revised articles from SELMAS 2003.

[102] Frédéric Souchon, Christelle Urtado, Sylvain Vauttier, and Christophe Dony. Exception handling in component-based systems: a first study. In A. Romanovsky, C. Dony, JL. Knudsen, and A. Tripathi, editors, *Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003*, pages 84–91, Darmstadt, Germany, July 2003. also available as Technical Report TR 03-028, Department of computer science, University of Minnesota, Minneapolis.

[103] Frédéric Souchon, Sylvain Vauttier, Christelle Urtado, and Christophe Dony. Fiabilité des applications multi-agents : le système de gestion d'exceptions SaGE. In Olivier Boissier and Zahia Guessoum, editors, *Systèmes multi-agents défis scientifiques et nouveaux usages – Actes des Journées Francophones sur les Systèmes Multi-Agents 2004*, pages 121–134. Hermès, Paris, France, November 2004.

[104] Petr Spacek, Christophe Dony, and Chouki Tibermacine. A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In *Procedings the 17th International ACM SIGSOFT Conference on Component Based Software Engineering*, pages 13—23, Lille, France, July 2014.

[105] Judith A. Stafford and Alexander L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 11(04):431–451, 2001.

[106] Eleni Stroulia and Yiqiao Wang. Structural and semantic matching for assessing web-service similarity. *International Journal of Cooperative Information Systems (IJCIS)*, 14(4):407–438, 2005.

[107] Clemens Szypersky, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming (Second Edition)*. Addison-Wesley / ACM Press, second edition, 2002.

[108] Guilaine Talens, Chabane Oussalah, and Marie-Fran¸oise. Colinas. Versions of simple and composite objects. In David A. Bell Rakesh Agrawal, Seán Baker, editor, *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB 1993)*, pages 62–72, Dublin, Ireland, August 1993. Morgan Kaufmann.

[109] Bruno Traverson. Abstract model of contract-based component assembly, 2003. ACCORD RNTL project deliverable number 4 (in french).

[110] Christelle Urtado and Chabane Oussalah. Complex entity versioning at two granularity levels. *Information Systems*, 23(2/3):197–216, 1998.

[111] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[112] Hans Van Vliet. *Software engineering: principles and practice*, volume 3. Wiley & Sons, third edition, 2008.

[113] W3C. Web services description language (wsdl) version 2.0 - part 1: Core language, 2007.

[114] Kurt C. Wallnau. Volume III: A technology for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon University, Pittsburgh, USA, April 2003.

[115] Yiqiao Wang and Eleni Stroulia. Semantic structure matching for assessing web-service similarity. In Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors, *Proceedings of the first International Conference on Service Oriented Computing (ICSOC 2003)*, volume 2910 of *LNCS*, pages 194—207, Trento, Italy, December 2003. Springer.

[116] William E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods*, pages 354—359. American Statistical Association, 1990.

[117] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In Muhammad Ali Babar and Ian Gorton, editors, *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, August 2010. Springer.

[118] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric development and evolution processes for component-based software. In *Proceedings of 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, pages 680–685, Redwood City, USA, July 2010.

[119] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Dedal : un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants. In Khalil Drira, editor, *Actes de la 4ème Conférence francophone sur les Architectures Logicielles (CAL 2010), Revue des Nouvelles Technologies de l'Information (RNTI L-5)*, pages 15–27, Pau, France, Mars 2010. Cépaduès.

[120] Huaxi (Yulin) Zhang, Christelle Urtado, Sylvain Vauttier, Lei Zhang, Marianne Huchard, and Bernard Coulette. Dedal-CDL: Modeling first-class architectural changes in Dedal. In *Proceedings of the Joint 10th Working IEEE / IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA / ECSA 2012)*, Helsinki, Finland, August 2012. Short paper. AR 35,7%. ERA rank A.

[121] Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. A three-level component model in component-based software development. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE 2012)*, Dresden, Germany, September 2012. ACM.

Appendices

Appendix A

Complex entity versioning at two granularity levels

This journal article [110] has been published in Information Systems in 1998. It is an extended and selected version of the conference paper [90] published at CAiSE in 1997. It is a good summary of my PhD thesis.

It defines a version management model tailored for what I called "complex entities" at that time. Entities because it is not only about versioning objects as class versions are considered too. Complex because it is not only about versioning isolated entities as dependancies between entities build entity dependency graphs in which versioning a node might in turn necessitate to version dependent ones. The proposed version management model, which is intended for application designers, manages evolution in an original parameterizable way:

- It divides into two user-levels: microscopic and macroscopic. Depending on their needs and on those of the targeted application, designers can choose the level they consider to suit them best. The microscopic level is intended for expert users who have specific version management needs, while the macroscopic level is much more suitable for non-expert users.
- It has not been defined for a particular category of dependence relations and is parameterizable: existential dependencies are dealt with by an operation propagation mechanism described with propagation rules and strategies.
- It is modular as designers can either reuse predefined rules and/or strategies or create new ones in order to respond to new requirements (new version propagation strategies, new dependency relation between entities, etc.).

Pergamon

Information Systems Vol. 23, No. 3/4, pp. 197–216, 1998

Printed in Great Britain
0306-4379/98 $17.00 + 0.00

COMPLEX ENTITY VERSIONING AT TWO GRANULARITY LEVELS

CHRISTELLE URTADO and CHABANE OUSSALAH

LGI2P / EMA - EERIE, Parc Scientifique Georges Besse, 30000 Nîmes, France

(Received 30 September 1997; in final revised form 12 February 1998)

Abstract — Engineering applications require semantically rich data to be modeled and managed over time. Complex entities, meaning entities linked to other entities by structural and existential dependences, allow such data to be represented. Complex entity versions track the evolution of complex entities over time or during a design process. This paper describes a complex entity version management model that combines two levels for the management of complex entity versions, depending on the granularity of the versioned entities: one macroscopic and the other microscopic. These two levels give the various categories of users the possibility to choose the complex entity version management technique best adapted to their needs. Unlike existing models, this model allows all types of complex entities, i.e. both classes and instances, to be managed. Furthermore, the microscopic level is highly parameterized and the macroscopic level offers an easy-to-use interface to users. Copyright ©1998 Elsevier Science Ltd

Key words: Entity Version Management, Complex Classes or Instances, Macroscopic and Microscopic Levels, Version Propagation Rules and Strategies.

1. INTRODUCTION AND PROBLEMATICS

Any application supporting a design or creation activity (such as computer-aided design frameworks, software engineering environments, multimedia composition environments, hypertext support environments, etc.) requires persistent storage enriched with powerful techniques for the representation and management of data. Historically, these applications firstly expressed their need for the insertion into databases of structures allowing semantically rich data to be represented. This requirement is particularly well fulfilled by object-oriented databases integrating the complex entity notion. These applications, because of their interactions with several types of users, are highly dynamic; it must be possible to modify either the structure and organization of the objects stored in the database or the objects themselves. Storage of several states of data (instances) and of data structures (classes) is therefore required. This need for data evolution management techniques is widely considered to be fulfilled using the notion of version [20], whether it be by databases from research laboratories (Encore [23, 26], Orion [4], Version Server [12]), databases from industry (ObjectStore [11], Versant [18]) or by software engineering environments [5, 8, 9, 22]. The necessity to enhance database systems with complex entity version management techniques emerges from these two requirements.

1.1. Elementary versus Complex Entities

An *elementary entity* is a class or an instance which is independent of other classes or instances stored in the database. On the contrary, a *complex entity* is a class or an instance with references to other classes or instances (called its constituents) in the same database. These references are called *dependence relations*. The composition relation [13, 17] is an example of a dependence relation. We can thus talk about composite and component entities. The signification of elementary and complex entity versions can be deduced from the definitions given for elementary and complex entities.

Complex entity version management is an extension of elementary entity version management, for which we have previously proposed a model [24]. In this paper, the characteristics required for an elementary entity version model (elementary instance or elementary class version creation, version history, version state, storage, version id, etc.) are not addressed. For more details on these matters, the reader can refer to [10, 12, 16].

Complex entity version management aims to solve problems which are typical of the complex entity notion: the management of structural or existential dependences between the elementary entity versions which constitute a complex entity version.

1.2. Problematics of Complex Entity Version Management: Dependences between Constituents

Structural Dependence The entities stored in a database are dependent from one another: they represent a coherent state of the modeled world. This structural dependence is expressed by means of relations (such as inheritance, composition, association and equivalence). When several versions of the stored entities coexist in the same database, the database is responsible for presenting to users versions of entities that "go together" [2]. The database must therefore identify groups of entity versions that represent a coherent state of the modeled world. Consequently, a complex entity version management model should provides structures which allow groups of coherent versions, i.e. complex entity version constituents, to be grouped together.

Existential Dependence The elementary entity versions constituting a given complex entity version are, as we have just seen, dependent on one another. This dependence is not just structural, but existential too. In other words, the creation or destruction of a constituent entity version can influence the creation or destruction of versions of other constituents of the same complex entity. These operations could be carried out by users but, as soon as the entities become numerous, this task becomes too complex to be dealt with manually. Consequently, it is important that operational mechanisms for the automatic creation and destruction of complex entity versions be provided.

As a conclusion, in addition to functionalities for elementary entity version management, a complex entity version management model must provide the user with:

- a representation model for groups of coherent elementary entity versions,
- a technique for the creation and destruction of complex entity versions.

1.3. Two Approaches for the Management of Complex Entity Versions

A number of models propose mechanisms for complex entity version management, even though they use different terminologies, such as schema versions, database versions or software configurations. These are generally tailored to a particular type of complex entity and a particular category of applications. Among existing models, we have identified two distinct approaches for the representation and management of complex entity versions. They differ in the granularity of the versioned entities:

- The first approach, which we call the *microscopic approach*, considers complex entity versions as elementary entity version graphs whose edges are dependence relations. It provides automatic management techniques for such graphs.
- The second approach, which we call the *macroscopic approach*, considers complex entity versions as versions of high granularity entities we call containers, and provide container version management techniques. These containers can, for example, be a configuration or a database, and themselves contain elementary entity versions.

These two approaches happen to be complementary in terms of the applications supported, the granularity of the data versioned and the user profiles they are tailored to correspond to. The first approach is adapted to CAD applications which frequently need to compare several design alternatives for a given entity and which include distinct versions of a given entity in a single design artifact. The second approach is closer to the needs of computer-aided software engineering applications, which model programs as black boxes and need to consider versions of these boxes, each containing a single version of each entity (the debugged version, the version that corresponds to the characteristics of the targeted hardware, etc.). Moreover, these two approaches correspond to version management at two user levels. The first is appropriate for users with demanding needs

in terms of version management (numerous types of dependence relations, necessity to parameterize version propagation, etc.) and are therefore designing a real version management policy for their application. The second corresponds to the needs of non-expert users. It provides them with version management techniques which are easy to use.

To illustrate these differences, let us suppose that the object considered by the user is a car. This car is described as a composite instance: it is composed of a body and an engine plus other components. The complex object car exists in two versions: a first version with a body painted in dark green and having a quoted value of 100,000 FF and a second version painted in a metallic color giving it a higher quoted value (105,000 FF).

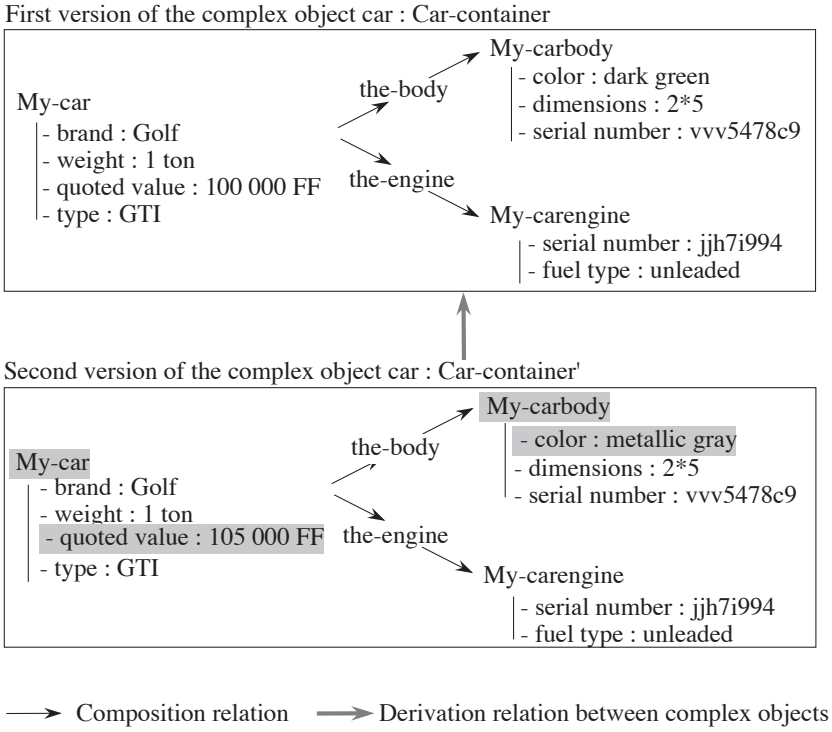


Fig. 1: Two Versions of the Complex Object Car at the Macroscopic Level

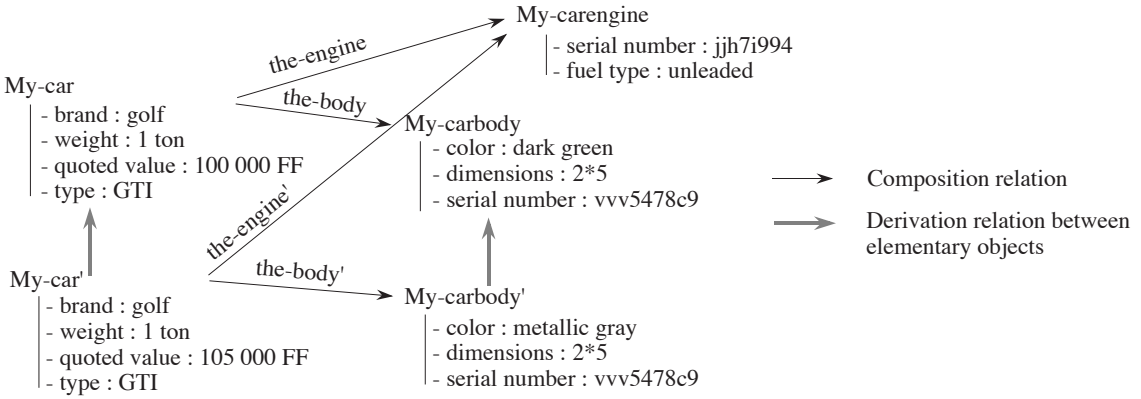


Fig. 2: Two Versions of the Complex Object Car at the Microscopic Level

At the macroscopic level, the user considers the complex object car as a box in which there is a version for the car (*My-car*), a version for the body (*My-carbody*) and a version for the engine (*My-carengine*), linked by composition relations (*the-body* and *the-engine*) as represented in the upper part of Figure 1. When he/she works with a given version of the complex object car, the physical representation of the various versions of the car's constituents is hidden from him/her.

At the microscopic level, the user considers each version of the complex object car as a composition graph (*cf.* Figure 2). He/she can identify the derivation hierarchies of the various constituents of the complex object and, for example, see that a version of the engine is shared by several versions of the car.

1.3.1. Problematics within the Framework of the Microscopic Approach

In this approach, the versioned entities are the entities having the lowest granularity of the database. Complex entity versions are seen as graphs which interconnect groups of coherent elementary entity versions (class versions or instance versions).

Many models use this approach for different types of complex entities, meaning different types of dependence relations. Let us cite: Version Server [12], Encore [26] and Présage [24] for the composition relation, Encore [23] for the inheritance relation, Version Server [12] for the equivalence relation, and Orion [4] for the simple referencing relation[†].

The two problems (structural and existential) which have been identified in Section 1.2 can be stated more precisely within the framework of the microscopic approach. In this approach, the problem of grouping sets of coherent elementary entity versions consists in maintaining dependence relations between elementary entity versions because complex entities and complex entity versions are seen as graphs. Most models support both static and dynamic referencing. Static referencing can involve the copying [1, 15, 23, 24] or the versioning [12] of the relation between entities. The creation or destruction of complex entity versions is generally carried out using a version propagation mechanism. "Propagation [...] is the automatic application of an operation to a network of objects when the operation is applied to some starting object" [21]. In the case of complex entity version management, this mechanism automatically creates or destroys a complex entity version by propagating the operation from a constituent to another through dependence relations. This technique makes the user's task easier when entity versions are too complex to be dealt with manually.

1.3.2. Problematics within the Framework of the Macroscopic Approach

The macroscopic approach consists in grouping together the elementary entities which constite the complex entity in a high-granularity container which is itself considered to be a versionable entity. The various versions of this container represent groups of coherent versions of its constituent entities. In the same way as we distinguish class versions from instance versions [24], we distinguish two types of containers:

- *class containers* which group together classes and class versions. The models that propose class container versions are, like the models that propose class versions, not very numerous because only a few models are interested in storing both object structures (classes) and objects themselves (instances). They are used by Kim and Chou [14] in their schema version management model and in VBD [3] where database versions contain class versions as well as instance versions.

- *instance containers* which group together instances and instance versions. This category of container is used by the VBD model [2] for database versions and by the ObjectStore system [11] for configuration versions[‡].

[†]The simple referencing relation does not have any predefined semantics, unlike, for example, the composition relation.

[‡]For ObjectStore, a configuration is a group of instances that has to be treated as a versioning unit.

The two problems (structural and existential) that have been identified in Section 1.2 can be stated more precisely within the framework of the macroscopic approach.

The need for complex entity version management models to maintain structural dependences between constituents of complex entities involves, in the case of the macroscopic approach, two sub-problems:

- *The Physical Description of a Container.* In the VBD model [2], an association table allows dynamically determining the instance version which belongs to a given database version. In Kim and Chou's model [14], each class version references its creator schema through a static link (attribute), while a corresponding generic object references the schema that has deleted a class. Their system can therefore dynamically determine which class version belongs to a given schema.

- *References Between Elementary Entity Versions.* In the VBD model, references are dynamic. If an entity A is linked through a simple referencing relation to an entity B, the entity version representing A in a database version is implicitly linked to the version representing B in this database version. In Kim and Chou's model, on the contrary, relations are static, encapsulated in classes, by means of attributes referencing superclasses and subclasses.

Existential dependence in complex entity version management involves, in the macroscopic approach, defining creation and destruction operations for a container version. In the VBD model, the creation of database versions is carried out by means of a special type of transaction. Therefore, the creation of a database version occurs at the user's initiative. In contrast, in Kim and Chou's model, the addition of a superclass or subclass to a given class automatically triggers the creation of a version of this class in the schema concerned.

1.4. Outline of the Paper

The main contribution of this paper is the definition of a complex entity version management model directed towards information systems that complements object-oriented database capabilities. This model is structured on two levels: an elementary entity version graph management technique (the microscopic level) and a container version management mechanism (the macroscopic level). Different types of users can use one or the other according to their needs and those of the targeted application. Technically, these two levels are based on the same basic notions (reified dependence relations, relation versions, propagation rules and strategies).

The remainder of this paper introduces our model for the management of complex entity versions. It is organized as follows. In Section 2, the basis of our model, to be used at both microscopic and macroscopic levels, is described. In Sections 3 and 4 respectively, the microscopic and macroscopic levels for complex entity version management are presented and illustrated. Section 5 presents our model from the user's point of view and details the proposed functionalities and the methodology to design an application's version management policy. Section 6 concludes the paper with perspectives for this work.

2. BASIS OF A MODEL USING A MIXED APPROACH

2.1. Objectives

Our aim is to provide the various types of user with a model for the management of complex entity versions which: is powerful, in that it applies to any category of application, expressive, in that it allows the user to define how complex entity versions are managed, and easy to use, especially by non-expert users. With this perspective in mind, we chose to adopt an approach combining the management of elementary entity version graphs (microscopic level) and the management of container versions (macroscopic level). Users therefore have at their disposal a range of possibilities which allows complex entity versioning to be dealt with at two distinct granularity levels.

The macroscopic level allows non-expert users to consider complex entity versions as black boxes which contain sets of coherent versions. The microscopic level and its associated version propagation mechanism allow application designers to establish a real complex entity version management policy that can adapt to the needs of an application. At the system level, these two approaches integrate very well because they are based on common concepts: the notions of dependence relation, relation version, propagation strategy and propagation rule.

Our complex entity version model is based on an existing representation model for elementary entity versions [24], extended to the representation of dependence relation versions, on top of which we have added three modules dedicated to complex entity version management: a module providing propagation rules and strategies, which defines basic concepts, and two modules for complex entity version management, one at the microscopic, the other at the macroscopic level.

2.2. Dependence Relations, Propagation Rules and Strategies

Relations In order to permit our model to be adapted to any type of complex entity, and to allow for addition of new types of relations later, we chose to reify references between entities, meaning to represent them using entities: *dependence relations*. These relations allow the definition of semantics for inter-entity references. In many models, the composition relation, for example, conveys strong semantics [13, 17]: a composition link can, for example, be exclusive or shared, dependent or independent, predominant or not. In order to uniformize the treatment of entities and relations, we decided that dependence relations were versionable. Therefore, as is the case for entities, dependence relations can be derived into *dependence relation versions*.

Propagation The way in which entity versions which are linked to a given entity are propagated is not a structural property of the entity: rather, it depends on the relation linking the entities. We therefore chose to attach the version propagation capabilities directly to the relations [21]. These version propagation capabilities are clustered into *propagation strategies*. A propagation strategy is therefore associated with each dependence relation, either directly or not: it can be inherited from a relation superclass or redefined[†]. Propagation strategies group together the set of *propagation rules* which define the version creation and version destruction propagation capabilities for the associated relations. Propagation rules define declaratively the actions that must be triggered on the entities which may be reached through the relation (that is, the extremities of the relation).

In order to simplify the use of propagation rules, the system possesses predefined rules adapted to the most current inter-entity relations. These rules are active rules: they are written with the formalism of ECA rules (Event / Condition / Action) [6]. They allow version creation and version destruction operations to be propagated in four directions and according to two modes [19, 25]. The *propagation direction* can be FORWARD, BACKWARD, BIDIRECTIONAL or NONE. FORWARD, for example, signifies that the propagation takes place from the source of the relation to its destination. The *propagation mode* can be RESTRICTED or EXTENDED. If it is RESTRICTED, the operation propagates from the extremity on which it is triggered to the relation entity. If it is EXTENDED, the operation propagates from the extremity on which it is triggered to both the relation entity and the other extremity of the relation.

Figure 3 illustrates the semantics of propagation rules. Let us imagine that rules R_3 and R_5 are associated to a dependence relation r linking an entity A to an entity B. Rule R_3 propagates version creation in the FORWARD direction in the EXTENDED mode. It signifies that, if a version of A is created, rule R_3 is responsible for propagating this operation by creating a version of r and a version of B and for the linking together of these new entity and relation versions. If B was linked to other entities, the creation of a version of B could, in turn, trigger other version creations. Rule R_5 propagates version creation in the BACKWARD direction in the RESTRICTED mode. It signifies that, if a version of B is created, rule R_5 is responsible for propagating this operation by creating a version for the dependence relation r and for the linking together of the new relation version, the new version of B and the entity A.

[†]For more details on the association of a propagation strategy to a relation, refer to section 3.2

Creation rule R3

```
direction : forward
mode : extended
event : X' <- create-version(X)
condition : let r(X, Y) be a relation
action : r' <- derive (r)
         Y' <- create-version (Y)
         r'.source <- X'
         r'.destination <- Y'
```

Creation rule R5

```
direction : backward
mode : restricted
event : Y' <- create-version(Y)
condition : let r(X, Y) be a relation
action : r' <- derive (r)
         r'.source <- X
         r'.destination <- Y'
```

Fig. 3: Two Version Creation Propagation Rules

3. THE MICROSCOPIC APPROACH

This approach, built upon the notions described in Section 2, is based on the representation of complex entities and complex entity versions by means of graphs expressing the dependence between entities or entity versions. From the operational point of view, the management of complex entity versions is ensured by an operation propagation mechanism.

3.1. Complex Entity Version Derivation

The derivation of a complex class version can be decomposed in terms of the following elementary transformations: addition, suppression or modification of an attribute of the class (which are operations usually defined for elementary class versions) or addition, suppression, or modification of a relation coming from (efferent) or going to (afferent) this class. These last operations are charcteristic of complex classes. The derivation of a complex instance version, however, is limited to the modification of an attribute of the instance (which is the operation usually defined for elementary instance versions) or the addition, suppression, or modification of a relation coming from (efferent) or going to (afferent) this instance, provided it respects the definition embedded in the instance's class (such as cardinality constraints).

3.2. A Parameterized Propagation Mechanism

The introduction of propagation rules and strategy allows our version propagation model to be highly customizable. Application designers can choose from a wide variety of version creation and version destruction propagation techniques depending on the needs of their application. These techniques, embodied in propagation strategies, can either be created from scratch, to meet the needs of an application, or reused. A new propagation strategy can be created by combining either existing propagation rules or newly created ones.

Our propagation model [19, 25] allows both class version propagation and instance version propagation as well as both version creation propagation and version destruction propagation. To do so, a propagation strategy references (*cf.* examples of strategies in Figure 7):

- rules for the propagation of the version creation operation (through the attribute *has-as-creation-rules*) and rules for the propagation of the version destruction operation (through the attribute *has-as-destruction-rules*),
- dependent relations for which the strategy governs class version propagation (through the attribute *is-the-default-strategy-for-class*) and dependent relations for which the strategy governs instance version propagation (through the attribute *is-the-default-strategy-for-instance*).

The default propagation strategy associated with a relation for class version propagation can be redefined for a subclass of this relation. The default propagation strategy associated with a relation for instance version propagation is defined at class level and can be redefined in a subclass of the relation or even in a relation instance.

Users can also change the strategy associated with a given relation, either at the class or instance level. This change is limited to a set of strategies which have been selected by its designer as being compatible with the relation semantics. We call this set the "set of admissible strategies" [19, 25]. This notion is represented, for both levels, in the propagation strategy by referencing a list of relations for which the strategy is admissible at class level (through the attribute *is-an-admissible-strategy-for-class*) and a list of relations for which the strategy is admissible at instance level (through the attribute *is-an-admissible-strategy-for-instance*). When changing the default strategy associated to a relation at either level, the user must choose an admissible one.

3.3. An Example of Complex Object Version Management

The following example is inspired by a real case study involving the design and construction of a new electric power plant by a power supply company [7]. The complex class represented in Figure 4 represents the CAD project management part which models both the equipment involved in the project (*Engineering*, *CADsystem*, *DrawingBoard* and *CADstation*) and the designed object (*Project*) as being a composite class composed of *Assembly*, itself composed of *Part*. This example involves three types of relations between classes : the inheritance relation (*Part-having-a-hole* inherits from *Part*), the composition relation (*CADsystem* is composed of *CADstation*) and the association relation (a *Drawing* represents a *Part*).

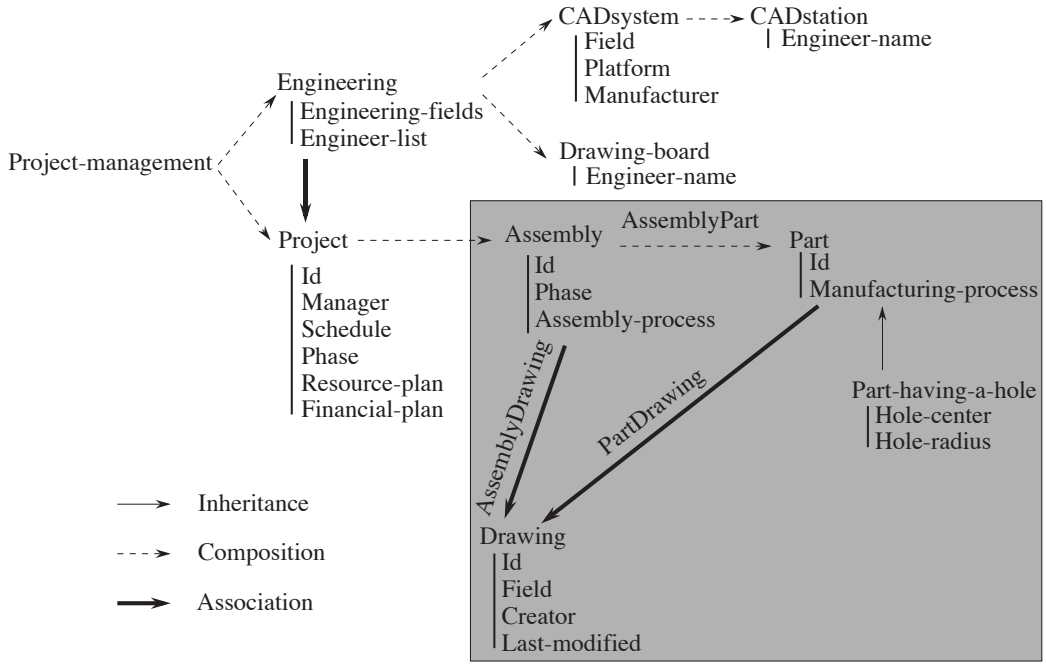


Fig. 4: Representation of the Complex Class Modeling the Design and Construction of an Electric Power Plant

We are considering a complex instance version management example based on this complex class representation. To enhance the readability of the example, we do not manage the whole instance graph but restrict ourselves to a part of the graph the classes of which are shaded in Figure 4. Let us consider, in the mechanical CAD domain, an assembly *a-101-102* which results from the bolting together of two parts each with a hole in it, *p-101* and *p-102*. Each one of these parts is represented by two drawings (*d-101-front* and *d-101-top* for *p-101* and *d-102-front* and *d-102-top* for *p-102*). The assembly is itself represented by a drawing *d-101-102-front*. The complex instance version for this example is represented in Figure 5.

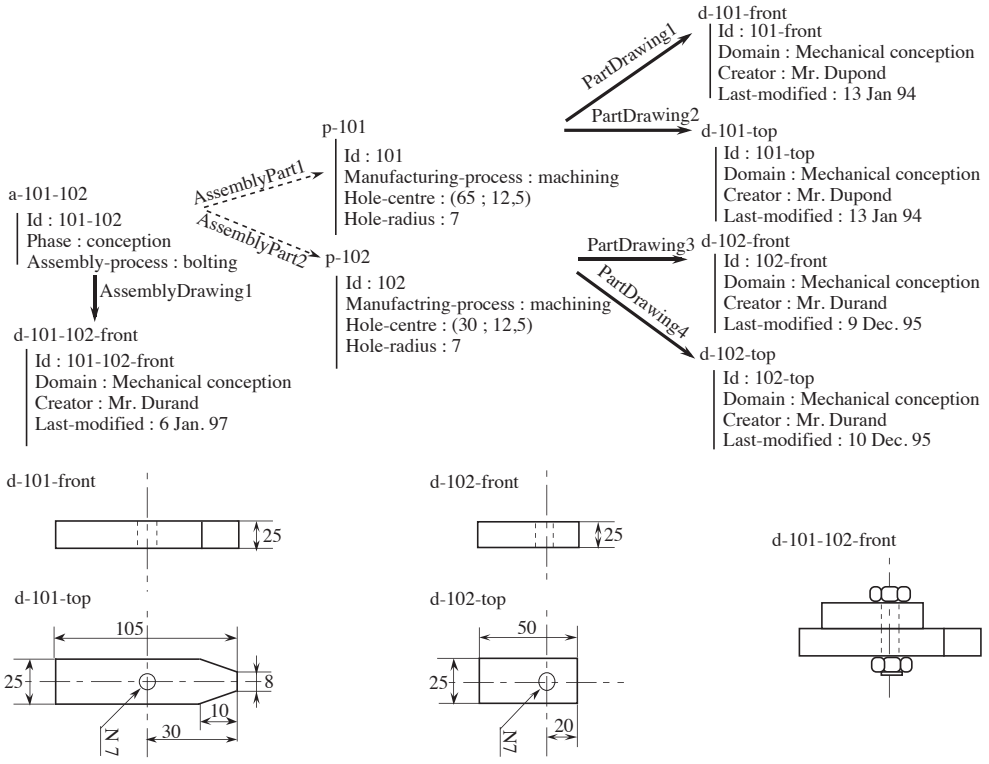


Fig. 5: Complex Instance Version Representing the Assembly, the Parts and the Associated Drawings

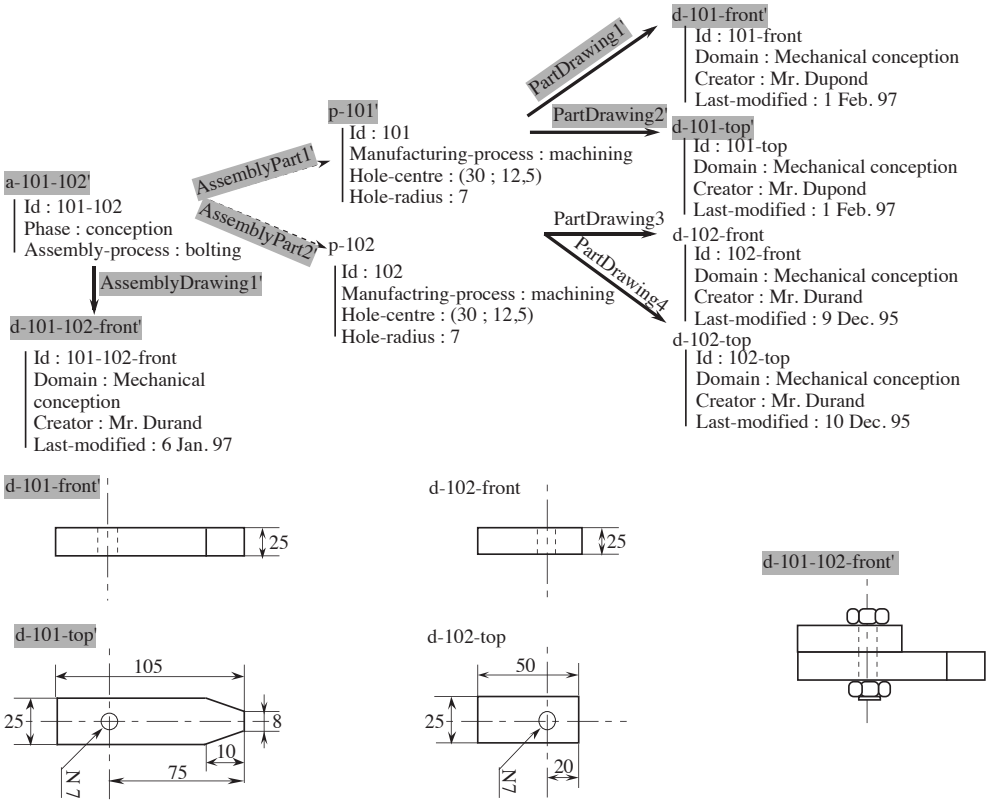


Fig. 6: New Complex Instance Version Representing the Assembly, the Parts and the Associated Drawings

Let us suppose that Mr. Dupond, designer of the front and top view of the part *p-101*, intends to modify the design of this part: he wants to move the centre of the hole. In order to visualize the impact of this modification on all the instances concerned , he derives a new version of the part *p-101*. Because of the rules governing technical drawing, these modifications will propagate to other instances. A version of each of the drawings *d-101-front* and *d-101-top* as well as of the two relations *PartDrawing1* and *PartDrawing2* will be created to reflect the change of the position of the hole. No modification is necessary for the part *p-102* itself or for the drawings representing it. In contrast, versions of the relation instance *AssemblyPart1*, of the assembly *a-101-102* and of the associated drawing *d-101-102-front* are necessary.

The new version of the complex instance representing the assembly, the parts and the associated drawings is represented in Figure 6. The instance versions and the relation instance versions that have been created are shaded. Indeed, one can see that the only versioned drawings are those associated with part *p-101* and with assembly *a-101-102*. Part *p-102* and its associated drawings (*d-102-front* and *d-102-top*) have not changed (they have not been derived).

Figure 7 proposes a description of the propagation rules and strategies that allow the new complex instance version to be obtained from the initial objects. Dependence relations, instances of the relation class *AssemblyPart*, are governed by strategy S_2 while dependence relation instances of the relation classes *PartDrawing* and *AssemblyDrawing* are governed by strategy S_3. It can be seen that strategy S_3 could be declared as the default propagation strategy for *AssemblyPart* or one of its instances because it is admissible for this relation class as an instance version propagation strategy. Strategies S_2 and S_3 respectively reference (R_2,R_4) and R_3 as their version creation propagation rules. These rules propagate version creation either in the FORWARD or in the BACKWARD direction and either in the EXTENDED or in the RESTRICTED mode.

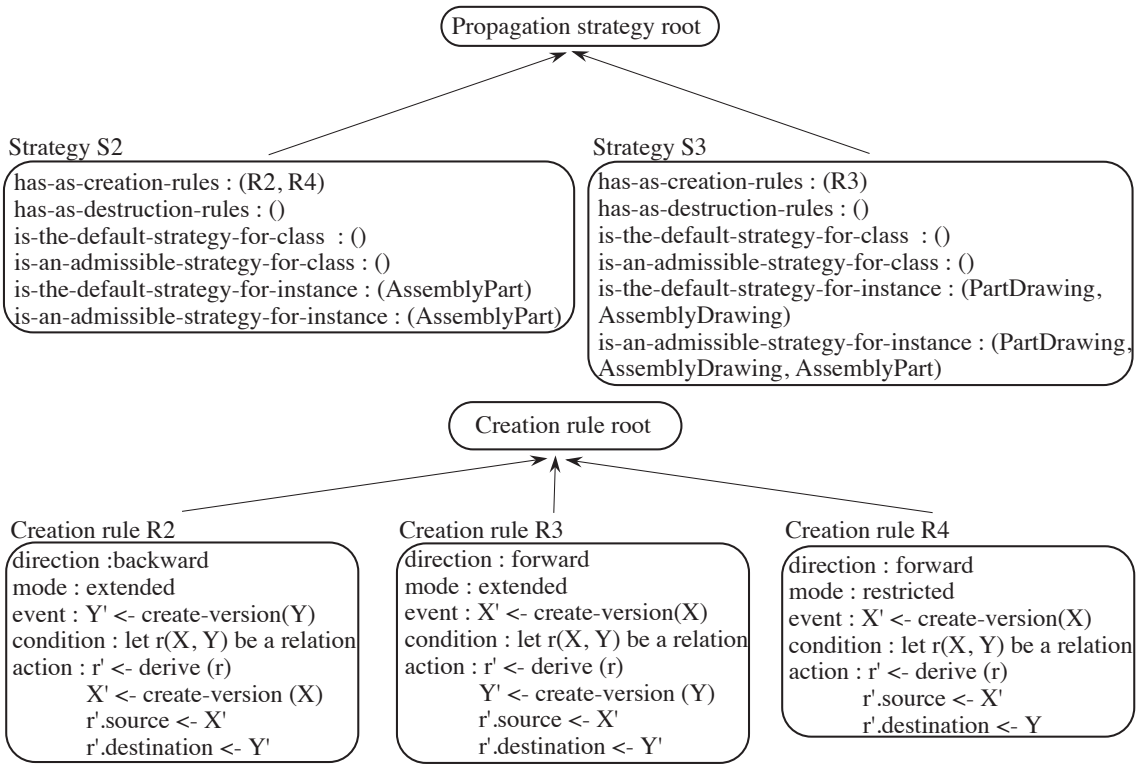


Fig. 7: Propagation Rules and Strategies Governing the Assembly, the Parts and the Associated Drawings

4. THE MACROSCOPIC APPROACH

The macroscopic approach is a technique for the simple expression of the most frequently needed complex entity version mechanisms. In the design of our complex entity version management model at the macroscopic level, we paid particular attention to the ease of use and flexibility of the notion of container (which is not limited, as in most of the existing models, to a single dependence relation fixed at the moment of the design of the framework of the model). Furthermore, unlike most of the existing models, our model allows both class-version and instance-version containers to be represented.

4.1. Container Creation

A container is a high granularity entity that groups together a set of elementary entities linked by dependence relations. A container is created by the user: it consists in enumerating all the elementary entities and relations of which it is constituted. A container can itself be constituted of other containers - which become sub-containers. This notion is similar to the notion of sub-configuration encountered in ObjectStore [11].

4.2. Container Version Creation

This section presents the container version derivation operation first from the use'sr point of view and then from the system's point of view.

4.2.1. User's Point of View

The creation (or derivation) of a container version respects the following principles:

- only one version of a given entity can be contained in a given container,
- containers are, in the same way as elementary entity versions, organized in a derivation tree with the predecessor/successor notion,
- if an entity version is not physically stored in a container version, it signifies that this entity keeps the value it had in the predecessor container version,
- when a container version is created, it is by default identical to its predecessor container version.

The derivation relation between containers allows the changes that have been made between two successive container versions to be tracked. Deriving a container version is associated with a combination of one or more of the following elementary transformations: addition, suppression or modification of one of its constituent entities. The modification of a constituent entity is equivalent to its derivation. The only operatiosn possible on instance containers are those respecting the semantics of container instantiation: modification of an existing constituentand possibly addition or suppression of a constituent provided this operation respects the dependence relations' description at class level (such as cardinality). When deriving a container, neither the addition, nor the suppression, nor the modification of a relation is permitted because we consider that containers define complex entities as wholes which consist of elementary entities and relations. The internal organization of entities inside a container is a characteristic of the container itself which is defined, once for all, by the user. Dependence relations and dependence relation versions are managed automatically by the system. Thus, the user's view of complex entity versions is simplified. When deriving a container version, the user only has to indicate which constituent entities (or entity versions) have to be added, suppressed or derived.

4.2.2. System Point of View

Macroscopic entity versioning is based on the concepts defined in Section 2 and already used by the microscopic approach. Our response to the structural dependence problem identified in Section 1.2 is based on the existing notions of dependence relation and relation version. However, these mechanisms are used only at the system level and remain hidden from users, in order to simplify the way in which macroscopic entity versions are presented to them. The creation (or destruction) of a container version is carried out by calling a version creation (or destruction) method which is distinct from the one defined for elementary entity version management. A container is linked to its constituent entities by a system-defined relation (the *is-contained-in* relation), with which a propagation strategy is associated in the same way as at the microscopic level. This strategy is hidden from users. It is composed of a single creation rule which is automatically generated by the system, given the list of the entities to add, suppress or derive, and of a single destruction rule. These system-managed strategies and rules cannot be modified by users: the system assumes the responsibility of maintaining the coherence of the containers. The execution of the container version creation rule or container version destruction rule follows the same principle as the one described at the microscopic level. However, the propagation rules executed here never trigger version creation recursively: they are written exclusively with terminal derivation operations that do not raise any event (except for the case of sub-containers).

4.3. An Example of Composite Object Version Management

There follows an example from the software engineering domain. The user's point of view is presented first. Then, the system's point of view is developed.

4.3.1. User's Point of View

Let us suppose that the object considered by the user of a software engineering application is a software pack for desktop publishing, modeled in order to be able to track its different versions for various platforms. This desktop publishing software pack is composed of a word processor, a spreadsheet and a drawing program. The word processor in turn is composed of a main module and a dictionary. The spreadsheet in turn is composed of a main module, a dictionary and a macro function interpreter. The drawing program is composed of a vector drawing module. Each of these instances possesses attributes, according to the description made in their respective classes, which, for simplicity's sake, we do not describe. This example illustrates our model for the composition relation but this choice is only exegetic: the example could very well have involved other types of relations (association, equivalence, etc.) and even combined them.

The container in question, *Software-container*, is an instance container. Its description consists of an enumeration of all the contained instances (*Software-pack*, *Word-processor*, *Spreadsheet*, *Drawing-program*, *Word-processor-main-module*, *French-dictionary*, *Spreadsheet-main-module*, *Vector-drawing-module*, *Macro-function-interpreter*) and of all the contained relations (*the-wp*, *the-spreadsheet*, *the-drawing*, *the-main-wp*, *the-dictionary-wp*, *The-dictionary-sprd*, *the-main-sprd*, *the-interpreter*, *the-main-drw*), among all the instances and all the relation instances available in the application. Figure 8 shows the container *Software-container*.

The user of the software engineering application needs to model a second version of the software, adapted to another platform (*cf.* Figure 9).

In order to express the modifications that must be made by the system to create a new version of the *Software-container*, the user must specify:

- the list of instances that have to be modified. In this example, this list is: (*Software-pack*, *Word-processor*, *Spreadsheet*, *Drawing -program*),
- the list of instances that have to be deleted. In this example, this list is: (*Macro-function-interpreter*),
- the list of instances that have to be added. In this example, this list is: (*English-dictionary*) as a component of *Word-processor* through the composition link *the-dictionary-wp*.

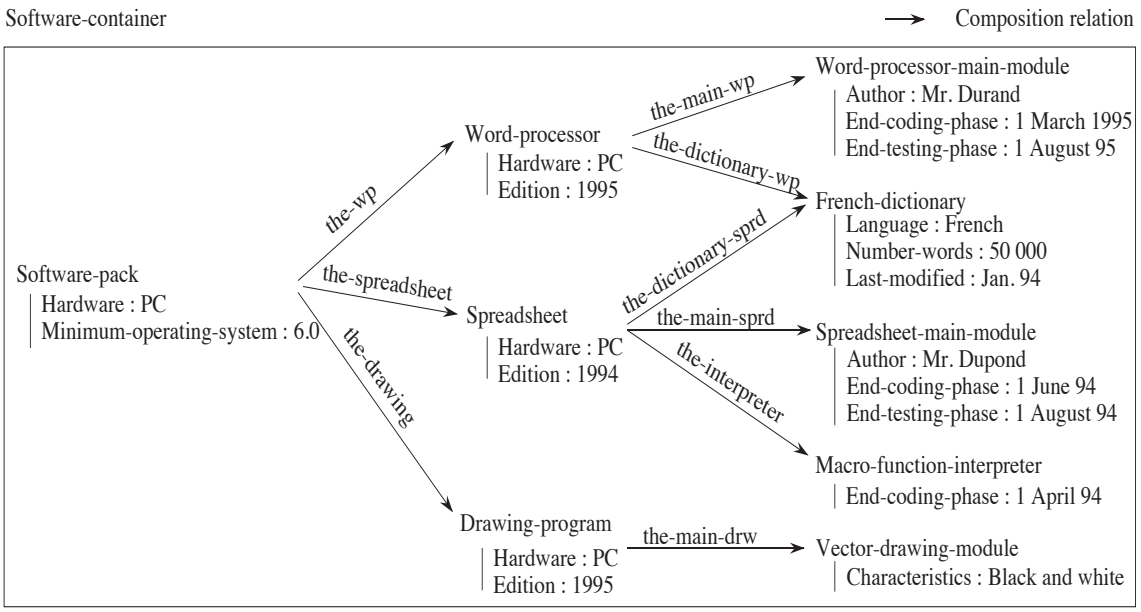


Fig. 8: A Version of the Instance Container Software-Container Representing a Desktop Publishing Software Pack

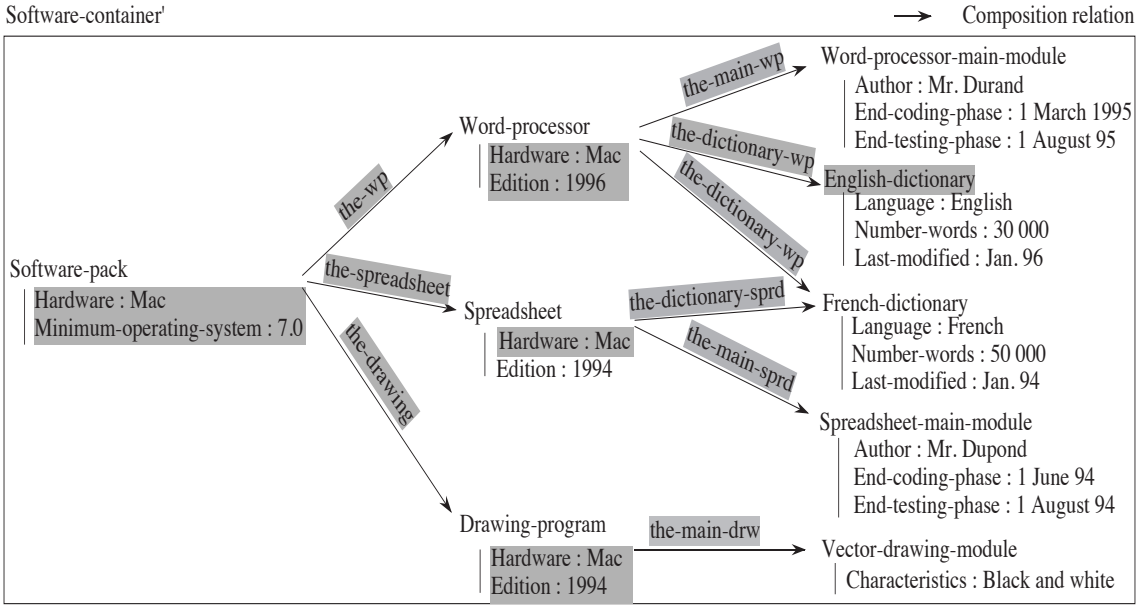


Fig. 9: The New Version of the Instance Container Software-Container

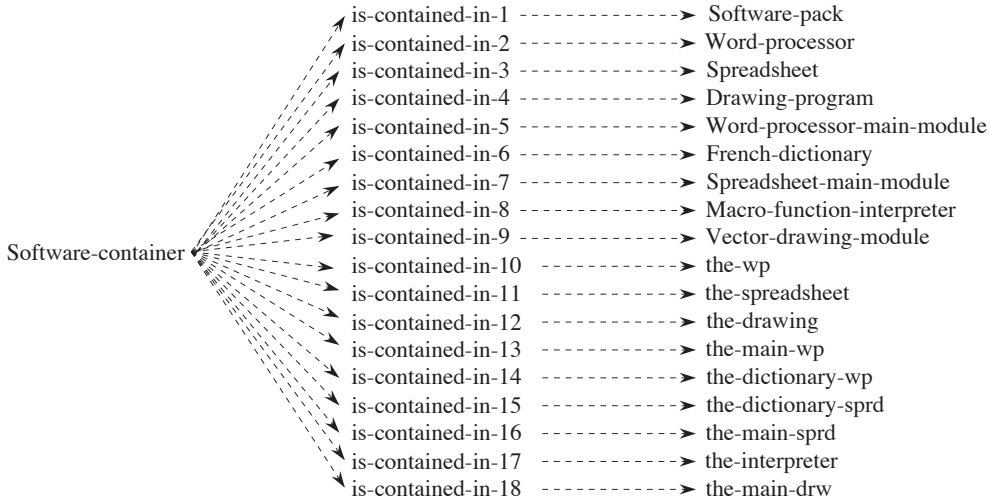


Fig. 10: System Representation for the Container Software-Container

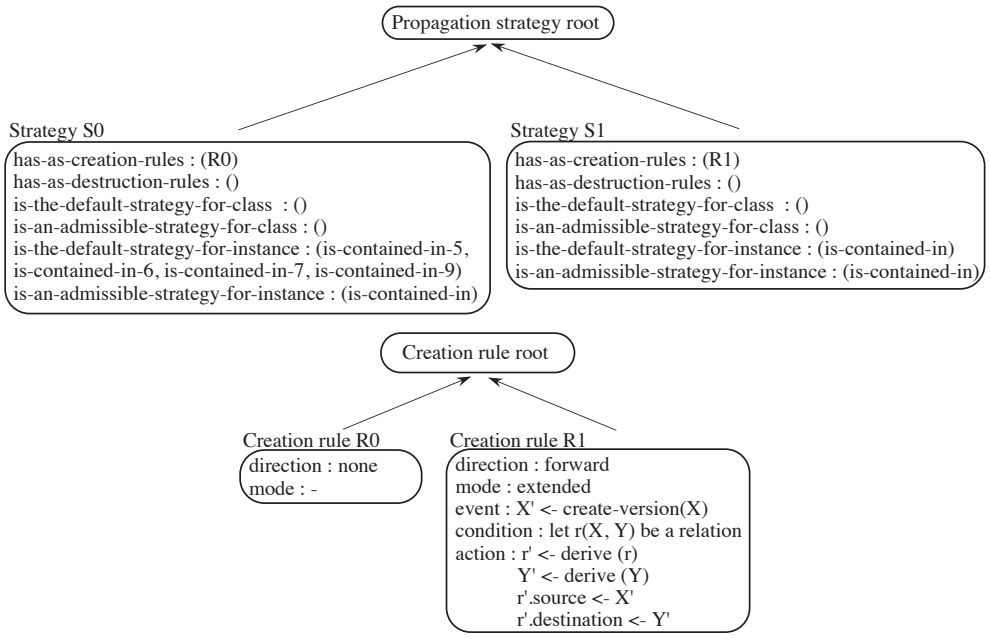


Fig. 11: Propagation Rules and Strategies Generated by the System for the Software-Container Example

4.3.2. System Point of View

At the system level, the container *Software-container* is represented as shown in Figure 10. To simplify the reading, neither the attributes, nor the arrows representing the composition hierarchy are given. The *is-contained-in* relation has been automatically introduced by the system and allows the instances and relation instances constituting the container to be identified. Moreover, it supports the version propagation strategies that allow the versions of *Software-pack*, *Word-processor*, *Spreadsheet*, *Drawing-program*, *the-wp*, *the-spreadsheet*, *the-drawing*, *the-main-wp*, *the-dictionary-wp*, *the-dictionary-sprd*, *the-main-sprd* and *the-main-drw* to be created. These strategies, as well as the corresponding propagation rules, are automatically generated by the system. The propagation rules and strategies generated for the *Software-container* are shown in Figure 11.

The version creation propagation rule for each one of these strategies is generated as follows:

- the rule propagates version creation in the FORWARD direction and with the EXTENDED mode for the system-managed relation *is-contained-in* referencing each object listed by the user as needing modification. In the *Software-container* example, the objects concerned are *Software-pack*, *Word-processor*, *Spreadsheet* and *Drawing-software*.

- the rule propagates version creation in the FORWARD direction and with the EXTENDED mode for the system-managed relation *is-contained-in* referencing each relation having an extremity among the objects listed by the user as needing modification. In the *Software-container* example, the relations concerned are *the-wp*, *the-spreadsheet*, *the-drawing*, *the-main-wp*, *the-dictionary-wp*, *the-dictionary-sprd*, *the-main-sprd* and *the-main-drw*.

- the rule propagates version creation in the FORWARD direction and with the EXTENDED mode for the system-managed relation *is-contained-in* referencing each sub-container listed by the user as needing modification. The rule is, in this case, responsible for the recursive propagation of version creation on these containers.

- the rule propagates version creation in the NONE direction for the system-managed relation *is-contained-in* referencing all other objects and relations. In the *Software-container* example, the objects concerned are *Word-processor-main-module*, *French-dictionary*, *Spreadsheet-main-module* and *Vector-drawing-module*. No relations are concerned.

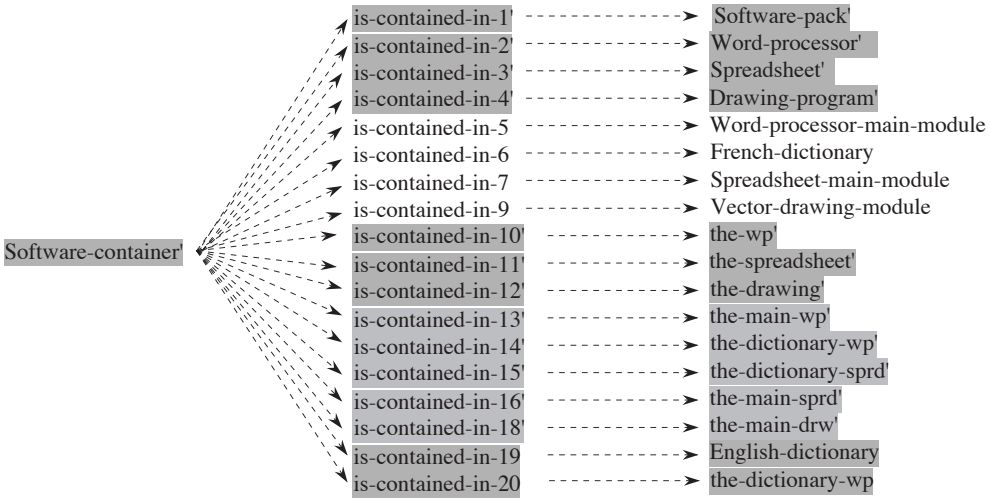


Fig. 12: System Representation of the Container Version Software-Container

Strategy S_1 governs version creation propagation for all instances of the *is-contained-in* relation. Strategy S_0 redefines the version propagation capabilities for the indicated instance relations, such as, for example, *is-contained-in-5*. This redefinition allows, for example, *Main-module-word-processor* not to be derived (rule R_0).

The version *Software-container'* of the container obtained after derivation is represented in Figure 12. The objects or relations that have been derived or added are shaded: for example, *Word-processor'* derives from *Word-processor* and *English-dictionary* has been added.

5. USE OF THE COMPLEX ENTITY VERSION MODEL

In this section, we present our model from the users' point of view: the two user categories, together with, on the one hand, the set of functionalities users have at their disposal to design the version management policy of their application, and, on the other hand, the execution of the complex entity version management mechanism. Our complex entity version model is currently being developed in C++ over the Versant [18] object oriented database in an application dedicated to network modeling assisted design. The possible application domains for this framework are: electronic circuit CAD, mobile telephone network modeling, etc.

5.1. The Two Categories of User: the Application Builder and the End-User

This framework is designed for two categories of users:

- the *application builder* (AB) is an expert in the application domain. He/she uses the complex entity version management model to design applications. The application builder is responsible for the definition of the classes of an application and their interdependencies. He/she therefore defines the version management policy for both class and instance levels;

- the *end-user* (EU) uses, at the instance level, the applications designed by the application builder. He/she does not define the version management policy. However, the end-user can modify this policy by choosing, for a given relation instance, a strategy which is different from the one proposed as a default strategy for instances at the relation class level from among the list of the admissible strategies for the relation instance.

The application builder works with classes that have been implemented as C++ instances of the framework classes (representing metaclasses). The information stored in these objects allows an automatic C++ code generation process to create "real" C++ classes that will be at the end-user's disposal for instantiation after a compilation phase. Moreover, at end-user level, an instance can access its class thanks to a particular instance we call the prototype instance.

5.2. Design of the Version Management Capabilities of an Application

In order to define the capabilities of his/her application's relations for version propagation, the user first has to choose between the macroscopic and the microscopic levels. This choice determines the set of functionalities he/she can use to define the version management policy for his/her application.

5.2.1. Microscopic Level

The application builder uses a language enabling him/her to design the version management policy of his/her application. He/she can:

- create new propagation rules.
  For this purpose, he/she can use one of the two constructors of the class *PropagationRule* either to create a rule from scratch or to re-use an existing rule class through inheritance.

- create new propagation strategies by combining predefined or newly created rules.
  For this purpose, he/she can use one of the two constructors of the class *PropagationStrategy* either to create a strategy from scratch or to re-use an existing strategy class through inheritance, as well as the methods *AddCreationRule* and *AddDestructionRule* allowing him/her to modify an existing strategy.

- choose the default strategy for a relation.
  For this purpose, he/she can use the methods *AddDefaultStrategyForClass* and *AddDefaultStrategyForInstance* from the class *PropagationStrategy* allowing him/her to achieve this operation for relation classes or relation instances.

- modify the list of admissible strategies associated with a relation.
  For this purpose, he/she can use the methods *AddAdmissibleStrategyForClass* and *AddAdmissibleStrategyForInstance* from the class *PropagationStrategy* allowing him/her to achieve this operation for relation classes or relation instances.

- change the default strategy for a relation by choosing an admissible strategy.
  For this purpose, he/she can use the methods *ChangeDefaultStrategyForClass* and *ChangeDefaultStrategyForInstance* from the class *PropagationStrategy* allowing him/her to achieve this operation for relation classes or relation instances.

- create new types of relations, for example by specializing existing ones.
  For this purpose, he/she can use the constructors of the class *Relation*.

In order to design the version management policy of his/her application, the application builder must conform to the following methodology:

Step 1:
Identify the application's relation classes. For each of these, determine if the predefined propagation strategy has the expected behavior. If it has, the AB has nothing to configure and his/her application's version management will be entirely automatic. If it has not, determine if an admissible strategy has the expected behavior. If there is one, go direct to step 5.
Step 2: There is no suitable admissible strategy.
Determine if one of the existing strategies has the expected behavior. If there is one, go to step 4.
Step 3: Creation of a strategy
Either re-use existing version creation and destruction propagation rules or create suitable new ones. Create a strategy by specifying the creation and destruction rules.
Step 4:
Declare that the strategy is admissible for the relation.
Step 5:
Set the strategy as the default propagation strategy for the relation.

The same methodology is to be followed for relation instances.

The end-user can change the default propagation strategy associated with a relation instance by choosing from among the set of admissible strategies for this relation. For this purpose, he/she can use the method *ChangeDefaultStrategyForInstance* from the class *PropagationStrategy*.

In order to describe the instance version management policy for the mechanical CAD example, the application builder executes C++ instructions through the graphical interface. Figure 13 is the retranscription of the generated C++ code corresponding to a scenario for the definition of the example's version management policy. This code corresponds to the execution of the successive steps proposed in the methodology. It is composed of a series of method-calls which pertain to the version management policy design language we have just described. In order to show the maximum number of steps, we have chosen to develop the scenario where no existing rule or strategy can be reused. In most of the cases however, the application builder would directly use the default strategies or execute steps 4 and 5 only.

5.2.2. Macroscopic Level

The interaction of users with the version management tool at the macroscopic level is extremely simple. The user just has to:

- define the container - that is enumerate all its constituent entities and relations. For this purpose, he/she can use the methods *AddConstituant* and *AddRelation* form the class *Container*.

- list the modifications the system must make to create a version of the container - that is enumerate the set of entities that must be modified, the set of entitites that must be suppressed and the set of entities that must be added. For this purpose, he/she can use the methods *DeriveWithModificationConstituent*, *DeriveWithSuppressionConstituent* and *DeriveWithAdditionConstituent* from the class *Container*.

```
// Step 3
// Creation of the version propagation rules
PropagationRule* R2 = new PropagationRule("backward", "extended", "Y' <- create-version(Y)", "let r(X, Y) be a relation", "r' <- derive (r); X' <- create-version (X); r'.source <- X'; r'.destination <- Y' ");
PropagationRule* R3 = new PropagationRule("forward", "extended", "X' <- create-version(X)", "let r(X, Y) be a relation", "r' <- derive (r); Y' <- create-version (Y); r'.source <- X'; r'.destination <- Y' ");
PropagationRule* R4 = new PropagationRule("forward", "restricted", "Y' <- create-version(Y)", "let r(X, Y) be a relation", "r' <- derive (r); r'.source <-X'; r'.destination <- Y' ");
// Creation of the version propagation strategies
List<PropagationRule*>* L1, L2;                          // creation of two empty propagation rule lists
L1 += R2;
L1 += R4;                                   // L1 contains version creation rules for the strategy being built
PropagationStrategy* S2 = new PropagationStrategy (L1, L2);          // L2, list of destruction rules, remains empty
L1->flush();                                                                                  // empties L1
L1 += R3;                                  // L1 contains the version creation rule for the strategy being built
PropagationStrategy* S3 = new PropagationStrategy (L1, L2);
// Step 4
S2-> AddAdmissibleStrategyForInstance (AssemblyPart);
S3-> AddAdmissibleStrategyForInstance (PartDrawing);
S3-> AddAdmissibleStrategyForInstance (AssemblyDrawing);
S3-> AddAdmissibleStrategyForInstance (AssemblyPart);
// Step 5
S2-> AddDefaultStrategyForInstance (AssemblyPart);
S3-> AddDefaultStrategyForInstance (PartDrawing);
S3-> AddDefaultStrategyForInstance (AssemblyDrawing);
```

Fig. 13: Design by the AB of the Version Management Policy of His/Her Application

5.3. Execution

The execution model is used for the implementation of both the microscopic and the macroscopic levels. This paragraph introduces the execution model and briefly explains how we implement the activity of propagation rules.

The code for the condition and action part of the version propagation rules is stored as text. It is composed of a series of message-calls for version creation (with or without modification of the newly created version) and for version destruction. The code is scanned and interpreted in order to send the appropriate messages. This technique allows application builders to dynamically describe and use new propagation rules.

Version creation and version destruction is initiated by users (application builders or end-users). Through the graphical interface of the model, they send a *create-version[...]()*[†] or *destroy-version()* message to an entity E of the application. This message is intercepted by the version manager, which is responsible for the implementation of the activity of rules. The version manager undertakes the following operations:

[†]The notation *create-verion[...]()* is used here to symbolize all version creation messages, including those which modify the newly created version.

- for all relations afferent to E, access the propagation strategy which manages its behavior and trigger the version propagation rules corresponding to the event, if they exist;

- for all relations efferent to E, access the propagation strategy which manages its behavior and trigger the version propagation rules corresponding to the event, if they exist.

The triggering of rules results in the execution of their action part. This execution can raise new version creation or destruction events that will be executed in the same way, and recursively propagate operations on versions. In order to avoid cycles in the activation of rules, the version manager stores the names of entities and relations that have been treated during a given propagation process. This prevents messages concerning the same entity or relation from being taken into account more than once.

Let us illustrate this by the description of the firing of rules in the mechanical CAD example. Creation of a version of part *p-101* triggers the execution of the rules corresponding to the version propagation strategies that are associated to the relation instances coming from (efferent) or going to (afferent) *p-101*. It corresponds to the firing of rules:

- R_3 from the strategy S_3 on relations *PartDrawing1* and *PartDrawing2*, instances of *PartDrawing* for which S_3 is the default strategy at the instance level. The execution of rule R_3 creates a version of each of these two relations and a version of the associated drawings (*d-101-front* and *d-101-top*).

- R_2 from the strategy S_2 on relation *AssemblyPart1*, instance of *AssemblyPart* for which S_2 is the default strategy at the instance level. The execution of rule R_2 creates a version of this relation and a version of the assembly (*a-101-102*).

- R_3 from the strategy S_3 on relation *AssemblyDrawing1*, instance of *AssemblyDrawing* for which S_3 is the default strategy at the instance level. The execution of rule R_3 creates a version of this relation and a version of the associated drawing (*d-101-102-front*).

- R_4 from the strategy S_2 on relation *AssemblyPart2*, instance of *AssemblyPart* for which S_2 is the default strategy at the instance level. The execution of rule R_4 creates a version of this relation which references the old version of the part *p-102*, which is not derived.

6. CONCLUSION AND PERSPECTIVES FOR THIS WORK

In this article, we have proposed a mixed model for the management of evolution in object bases by means of versioning. This model manages complex entity versions at two levels: the macroscopic and microscopic. In this, it differs from the existing models that manage either elementary entity versions or versions of high granularity entities.

In designing this model, particular attention was paid to its extensibility and, especially, to the management of complex entity versions of any type. This is why both levels of our model are based on dependence relations between entities, relations for which the users can define new types, and on propagation rules and strategies, which constitute a declarative support for the description of the operational capabilities of the dependence relations.

Perspectives for this work are numerous. The validation of our model on top of the database Versant [18] in its C++ version is in progress. We especially focus on the definition of an easy-to-use graphic interface for users of both levels of our model. We also plan to reinforce the links between class versions and instance versions by allowing the application designer to define versioning constraints on classes or class containers which are going to condition the versioning of instances or instance containers. Such constraints would allow quality criteria for end-user versions to be expressed.

REFERENCES

[1] D. Beech and B. Mahbod. Generalized version control in an object-oriented database. In *Proc. of the 4th IEEE Int. Conf. on Data Engineering*, pp. 14–22, Los Angeles, California (1988).

[2] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In *Proc. of the 16th Int. Conf. on Very Large Data Bases*, pp. 432–441, Brisbane, Australia (1990).

[3] W. Cellary, G. Jomier, and T. Koszlajda. Formal model of an object-oriented database with versioned objects and schema. In *Proc. of the 2nd Int. Conf. on Database and Expert Systems Applications*, pp. 239–244, Berlin, Germany (1991).

[4] H.T. Chou and W. Kim. Versions and change notification in an object-oriented database system. In *Proc. of the 25th ACM/IEEE Design Automaton Conference*, pp. 275–281, Anaheim (1988).

[5] S.A. Dart, R.J. Ellison, P.H. Feiler, and A.N. Habermann. Software development environments. *IEEE Computer*, **20**(11):18–28 (1987).

[6] U. Dayal, A.P. Buchman, and D.R. McCarthy. Rules are objects too : a knowledge model for an active, object-oriented database system. In *Proc. of the 2nd Workshop on Object-Oriented Database Systems*, pp. 129–143, West Germany (1988).

[7] D. Dori. Automated understanding of engineering drawings: An object-oriented analysis. *Journal of Object-Oriented Programming*, pp. 35–43 (1994).

[8] J. Estublier and R. Casallas. The Adele configuration manager. In *Configuration Management*, Trends in Software, chapter 4, pp. 99–133, J. Wiley & Sons (1994).

[9] A. Fugetta. A classification of case technology. *IEEE Computer*, **26**(12):25–38 (1993).

[10] D. Hsieh. Comparing the version management features of commercial ODBMSs. *Object Magazine*, pp. 65–74 (1994).

[11] Object Design Inc. Version management. In *ObjectStore Release 3 for UNIX Systems User Guide*, pp. 208–247 (1993).

[12] R.H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing surveys*, **22**(4):375–408 (1990).

[13] W. Kim, E. Bertino, and J.F. Garza. Composite objects revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Portland, Oregon (1989).

[14] W. Kim and H.T. Chou. Versions of schema for object-oriented databases. In *Proc. of the 14th Int. Conf. on Very Large Data Bases*, pp. 148–160, Los Angeles, California (1988).

[15] G.S. Landis. Design evolution and history in an object-oriented CAD/CAM database. In *Proc. of the IEEE COMPCON Conference*, pp. 297–303, San Francisco, California (1986).

[16] M.E.S. Loomis. Object versioning. *Journal of Object-Oriented Programming* (1992).

[17] M. Magnan. Reusability of components : exceptions in composite objects. PhD Thesis, University of Montpellier II, Montpellier, France (1994).

[18] Versant OBMS. *Versant Concepts and Usage Manual, Release 4.0* (1995).

[19] C. Oussalah and C. Urtado. Adding semantics for version propagation in OODBs. In *Proc. of the Workshop on Databases and Expert Systems Applications*, Zürich, Switzerland. IEEE Computer Society Press (1996).

[20] M.J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, **1**(4):364–370 (1975).

[21] J. Rumbaugh. Controlling propagation of operations using attributes on relations. In *Proc. of the Int. Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp. 285–296 (1988).

[22] B.R. Schmerl and C.D. Marlin. Versioning and consistency for dynamically composed configurations. In *Proc. of the 7th Int. workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pp. 49–65, Springer (1997).

[23] A.H. Skarra and S.B. Zdonik. The management of changing types in an object-oriented database. In *Proc. of the Int. Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp. 483–495 (1986).

[24] G. Talens, C. Oussalah, and M.F. Colinas. Versions of simple and composite objects. In *Proc. of the 19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland (1993).

[25] C. Urtado and C. Oussalah. Semantic rules to propagate versions in object-oriented databases. In *Proc. of the 3rd Workshop on Advances in Databases and Information Systems*, Moscow, Russia (1996).

[26] S.B. Zdonik. Object management system concepts. In *Proc. of the Conf. on Office Information Systems*, pp. 13–19, Toronto, Canada, ACM SIGOA (1984).

Appendix B

Search-based many-to-one component substitution

This journal article [42] has been published in a special issue on Search-Based Software Engineering of the Journal of Software Maintenance and Evolution: Research and Practice, in 2008. It presents part of the work realized during the PhD of Nicolas Desnos and with the collaboration of Guy Tremblay from Canada.

It presents a search-based automatic many-to-one component substitution mechanism. When a component is removed from an architecture to overcome component obsolescence, failure or unavailability, most existing fault tolerant systems try to find another comparable component to perform component-to-component (one-to-one) substitution. Doing so, they can only handle situations where a specific candidate component is available. As this is not the most frequent case, it would be more flexible to allow a single component to be replaced by a whole component assembly (many-to-one component substitution).

This article proposes such an automatic substitution mechanism, which does not require the possible changes to be anticipated and which preserves the quality of the assembly. This mechanism requires components to be enhanced with ports, providing synthetic information on their assembling capabilities. Such port-enhanced components then constitute input data for a search-based mechanism that looks for possible assemblies inside a component repository and progressively builds coherent architectures, using various heuristics to tame complexity.

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **00**:1–18 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Search-based many-to-one component substitution

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

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

SUMMARY

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

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

Introduction

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

*Correspondence to: LGI2P, Ecole des Mines d'Alès, Parc scientifique Georges Besse, F30035 Nîmes cedex, France
†E-mail: Nicolas.Desnos@ema.fr, huchard@lirmm.fr, tremblay.guy@uqam.ca, Christelle.Urtado@ema.fr, Sylvain.Vauttier@ema.fr

Received 31 August 2007
Revised 28 February 2008
Accepted 17 June 2008

Component-based software engineering (CBSE) [1] is a good solution to optimize software reuse and dynamic evolution while guaranteeing the quality of the software. Typically, a component is seen as a black box which provides and requires services through its interfaces. An architecture is built to fulfill a set of functional objectives (its functional requirements) while enforcing a set of properties (its non-functional requirements) and is described as a static interconnection of software component classes. A component assembly is a runtime instantiation of an architecture composed of linked component instances.

In long life applications or evolving environments, component substitution is a necessary mechanism for software evolution: it is a response to such events as component obsolescence, failure or unavailability. Anticipating valid component substitutions while designing some software is not always possible as the various contexts in which that software may run are not known in advance. Repairing a component assembly after a component has been removed while still preserving its whole set of functionalities is thus difficult. When a component is removed from an assembly, most existing approaches perform component-to-component (one-to-one) substitution [2, 3, 4, 5]. However, these approaches rely on the fact that an appropriate component, candidate for substitution, is available. This situation can hardly happen because it is difficult to find a component that has the same capabilities as the removed one. When such a component does not exist, allowing a single component to be replaced by a whole component assembly would permit more flexibility.

In this article, we propose an automatic substitution mechanism such that the possible changes do not need to be anticipated. Our approach uses primitive and composite ports for ensuring that a component can be replaced by a group of components while preserving the quality of the whole assembly. Such many-to-one component replacements are allowed by a search-based building algorithm that combines backtracking and branch and bound techniques to examine candidate assemblies. This algorithm is optimized using various search strategies and heuristics.

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

Software Architecture Correctness and Completeness in CBSE

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

Typical CBSE process. CBSE [7] makes it possible to build large systems by assembling reusable components. The life cycle of a component-based architecture can be divided into three phases: design-time, deployment-time and runtime.

At design-time, the system is analyzed, designed and the design validity is checked. An architecture is built to fulfill a set of functional objectives (its functional requirements) [8, 9]. Functional objectives are deduced from problem analysis and defined as a set of functionalities to be executed on selected components. Then, the structure of the architecture is described as a static interconnection of software component classes through their interfaces. This requires both selecting and connecting the software components to be reused[†]. This description, typically written in an architecture description language [10], expresses both the functional and non-functional capabilities of the architecture, as well as both the structural and behavioral dependencies between components. For simplicity's sake, this work only focuses on preserving functional requirements while the software evolves. Non-functional properties are also important but can be handled only after the functional constraints have been satisfied. Once the architecture is described, its validity is statically checked. Most systems verify the correctness of the architecture, while some also guarantee its completeness—both notions are described below. Once the validity of the architecture is checked, it can be deployed. Deployment requires instantiating the architecture, configuring its physical execution context and dispatching the components in this physical context. One of the results of deployment is a component assembly: a set of linked component instances that conforms to the architectural description. At runtime, the component assembly executes.

The evolution of this assembly is an important issue for the application to adapt to its environment in situations such as maintenance, evolution of the requirements, fault-tolerance, component unavailability, etc. In this context, an important question is: What are the possible dynamic evolutions that can be supported by the component assembly and by the architecture itself? The remaining of this paper is a tentative answer to this question.

Correctness. Verifying the correctness of an architecture amounts to verifying the connections between components and checking whether they correspond to a possible collaboration [9]. These verifications use various kinds of meta-information (types, protocols, assertions, etc.) associated with various structures (interfaces, contracts, ports, etc.). The most precise checks are done by protocol comparison, which is a hard combinatorial problem [11, 12, 13, 14, 15].

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

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

of an interface) can require the use of other functionalities which, in turn, entail new interface connections. Such functionalities (or interfaces) are said to be **dependent**. This information is captured in the description of component behavior and depends on the context in which the functionality is called (execution scenario). There are various ways to ensure completeness:

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

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

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

Dynamic Architecture Reconfiguration

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

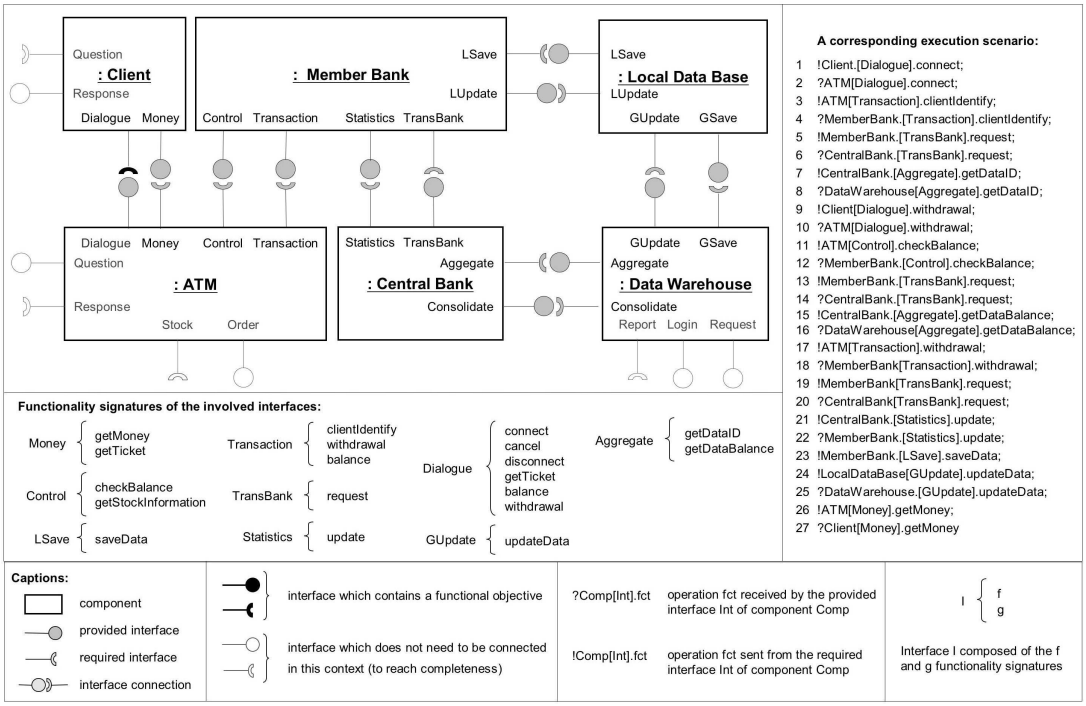


Figure 1. A complete assembly and a possible corresponding execution scenario

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

Dynamic Component Removal. Among the situations that must be handled to enable component assembly evolution is dynamic component removal. Other facets of interest in the

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

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

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

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

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

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

Port Enhanced Components for Incremental Assembly Completeness Checking

This section describes how adding ports to components provides a means to automatically build complete component assemblies [6, 28]. Existing approaches usually statically describe architectures in a top-down manner. Once the architecture is defined, they verify its validity using costly checking algorithms [11, 12, 13, 14, 15]. Our building of assemblies from components obeys an iterative (bottom-up) process. This makes the combinatorial cost of these algorithms critical and prevents us from using them repeatedly, as a naive approach would require. To reduce the complexity, we chose to simplify the information contained in protocols—more precisely, behavior protocols such as those used in SOFA [29], which are regular expressions that express the various possible sequences of events (traces) allowed by a component—and represent this information in a more *abstract* and usable manner through primitive and composite ports. Ports allow us to build a set of interesting complete assemblies from which it is possible to choose and check the ones that are best adapted to the architect's needs.

Primitive and Composite Ports. The underlying idea for building a complete component assembly is to start from the functional objectives, select the suitable components, and then establish the necessary links between them. Completeness is a global property that we will guarantee locally, in an incremental way, all along the building process. The local issue is to determine which interfaces to connect and where (to which component) to connect them. This information is hidden into behavior protocols where it is difficult to exploit in an incremental assembly process. We thus enhance the component model with the notion of port, to model the information that is strictly necessary to guarantee completeness in an abstract way. Primitive and composite ports will represent two kinds of connection constraints on interfaces, so that the necessary connections can be correctly determined. In some way, ports express the different usage contexts of a component, making it possible to connect only the interfaces which are useful for completeness. Primitive ports are used to model simple usage contexts (possible collaboration between two components) and composed into composite ports that model more complex contexts (complex collaborations). As in UML 2.0 [30], one can also consider that the functional objectives of an architecture are represented by use cases, that collaborations concretely realize use cases and contain several entities that each play a precise role in the collaboration. Primitive and composite ports can be considered as the part of a component that enables the component to play a precise role with respect to a given use case.

Primitive ports are composed of interfaces, as in many other component models [30, 31]. Ports are introduced as a kind of structural meta-information, complementary to interfaces, that group together the interfaces of a component corresponding to a given usage context. More precisely, a primitive port can be considered as the expression of a constraint to connect a set of interfaces both at the same time and to a unique component.

In Figure 2, the *Money_Dialogue* primitive port gathers the *Dialogue* and the *Money* interfaces from the *Client* component. It expresses a particular usage context for this component: here, a collaboration the *Client* component can establish with another (yet unknown) component to withdraw money. Connecting two primitive ports is an atomic operation that connects their interfaces: two primitive ports are connected together when all

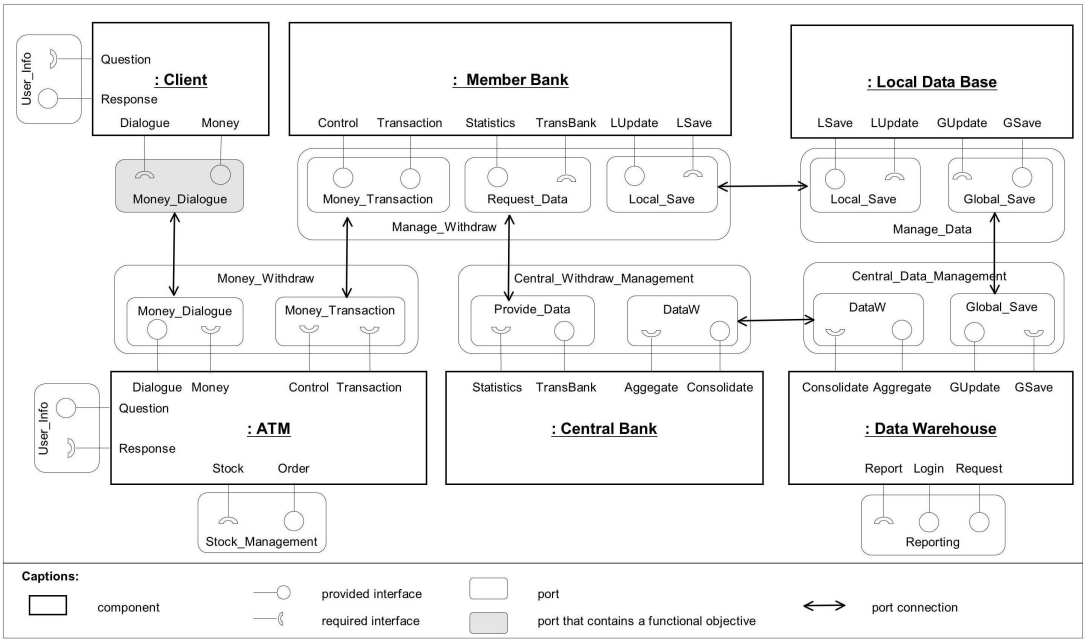


Figure 2. Example of components with primitive and composite ports

the interfaces of the first port are connected to interfaces of the second port (and reciprocally). Thus, port connections make the building process more abstract (port-to-port connections) and more efficient (no useless connections). In this example, the *Money_Dialogue* primitive port from the *Client* component is connected to the *Money_Dialogue* primitive port from the *ATM* component.

Composite ports are composed of other ports. They model complex collaborations that are composed of finer grained ones (modeled by the sub-ports). Indeed, they provide an abstract description of a part of the component behavior—less information than in component behavior protocols but more information than the syntactic description of component capabiliies modelled by interfaces. A composite port expresses a constraint to connect a set of interfaces at the same time but possibly to different components. In Figure 2, the *ATM* component has a composite port named *Money_Withdraw* which is composed of the *Money_Dialogue* and *Money_Transaction* primitive ports.

Much like a designer must do with protocols, ports have to be manually added to document the design of components; however, we are currently working on their automatic generation from behavior protocols.

Completeness of an Assembly as Local Coherence of its Components. Calculating the completeness of an already built component assembly is of no interest in an incremental building approach. Our idea is to better consider a local property of components that, if true, guarantees that the component is adequately connected to its immediate neighbors, and then to aggregate these local values into a global completeness property. We call this local property **coherence** and have shown [6] that it is a necessary condition for validity. Intuitively, we will see that when all components of an assembly are coherent, the assembly is complete. A component is said to be coherent if all its exposed (top-level) composite ports are and these latter are coherent if their primitive ports are connected in a coherent way (see below).

To determine the completeness of an assembly, we need to know if the interfaces that must be connected are indeed connected. The main idea is to check the coherence of each composite port. Two cases must be checked: when the composite port does not share any primitive port with another unrelated composite port and when it does share some primitive ports.

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

1. All its primitive ports are connected.
2. None of its primitive ports is connected.
3. Some, but not all, of its primitive ports are connected. In this case, the composite port can still be coherent if it shares the connected primitive ports with another unrelated composite port (of the same component) which is itself entirely connected. Indeed, sharing of primitive ports represents alternative connection possibilities [6]. A partially connected composite port can represent a role which is useless for the assembly as long as its shared primitive ports are connected in the context of another (significant) composite port.

A component is said to be coherent if all its exposed composite ports are coherent. An assembly of components is said to be complete if *i*) all the primitive ports which represent functional objectives are connected; *ii*) all its components are coherent.

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

Building Complete Component Assemblies: a Search-Based Problem

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

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

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

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

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

$$\begin{aligned} PrimPorts^*(\gamma) &= \{\rho \in \gamma \cap Prim_C\} \cup \bigcup_{\gamma' \in \gamma \cap Comp_C} PrimPorts^*(\gamma') \\ CompPorts^*(\gamma) &= \{\gamma' \in \gamma \cap Comp_C\} \cup \bigcup_{\gamma' \in \gamma \cap Comp_C} CompPorts^*(\gamma') \end{aligned}$$

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

$$\forall \gamma \in Comp_C \cdot \gamma \in TopComp_C \ \vee \ \exists \gamma' \in TopComp_C \cdot \gamma \in CompPorts^*(\gamma')$$

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

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

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

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

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

- We denote by $\widehat{\rho}$ the fact that, with respect to a set of components, ρ is *connected*—i.e., any required (resp. provided) interface of ρ is correctly linked with a provided (resp. required) interface of another (primitive) port.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

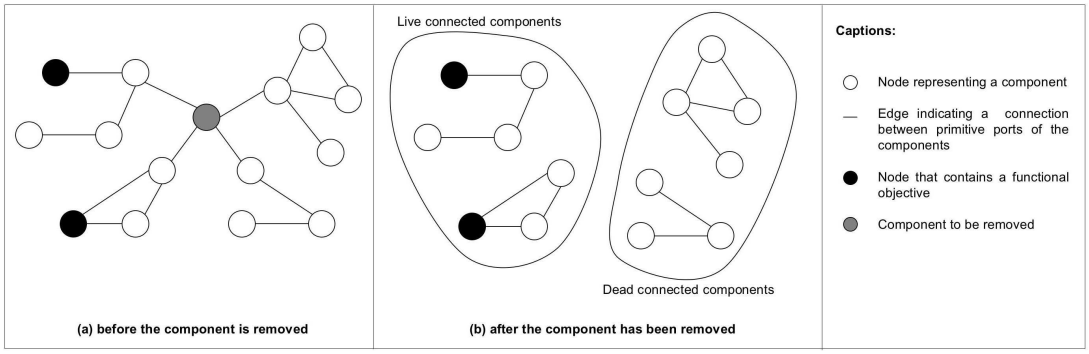


Figure 3. An assembly can be seen as (a) an abstract graph which is divided (b) in two sets of connected components when a component has been removed

Many-to-one Component Substitution Using the Automatic Building Process

To react to the dynamic removal of a software component, we propose a two step process that allows a flexible replacement of the missing component:

1. Analyze the assembly from which the component has been removed and remove the now useless (dead) components;
2. Consider the incomplete component assembly as an intermediate result of our iterative building algorithm and therefore run the building algorithm on this incomplete assembly to re-build a complete assembly.

Removing Dead Components. When a component has been removed from a complete assembly, some parts of the assembly may become useless. Indeed, some of the components and connections in the original assembly might have been there to fulfill needs of the removed component. To determine which parts of the assembly have become useless, let us define a graph which provides an abstract view of the assembly.

An assembly can be represented as a graph where each node represents a component and each edge represents a connection between two (primitive) ports of two of its components. We also distinguish two kinds of components: those which fulfill a functional objective—i.e., the components which contain a port which contains an interface which contains a functional objective—and those which do not (cf. Figure 3).

An **assembly** A can then be seen as a graph along with a set of functional objectives:

$$A = (G_A, FO_A)$$

Here, $G_A = (Cmps_A, Conns_A)$ is a graph, with $Cmps_A$ the set of nodes—each node being a component—, $Conns_A$ the set of edges—each edge indicating the existence of some primitive

port connection between the components—, and $FO_A \subseteq \bigcup_{C \in Cmps_A} Prim_C$ the set of primitive ports that contain some functional objectives[¶].

If we consider the graph that results from the removal of the node representing the removed component, we can partition the graph in two parts: the connected components[‖] that have at least a node which contains a functional objective and the connected components without any node that contains a functional objective. The second part of the graph is no longer useful because the associated components were not in the assembly to fulfill some functional objectives but rather to fulfill some of the removed component's needs. Removing this part of the graph amounts to removing now useless parts of the assembly before trying to re-build the missing part with new components and connections.

Let $A = (G_A, FO_A)$ be an assembly and let $C \in Cmps_A$ be the component to remove. We define $H_{A,C}$ as the graph G_A from which we removed component C and all the edges (denoted by $Conns_C$) corresponding to primitive port connections between C and another component of G_A:

$$H_{A,C} = (Cmps_A \setminus \{C\}, Conns_A \setminus Conns_C)$$

We define $\mathcal{L}_{A,C}$ the live connected components of $H_{A,C}$ as the graph composed of all the connected components of $H_{A,C}$ that have at least a node which contains a functional objective.

We also define $\mathcal{D}_{A,C}$ the dead connected components of $H_{A,C}$ as the graph composed of all the connected components of $H_{A,C}$ that have no node which contains a functional objective.

Let us just notice that:

$$H_{A,C} = \mathcal{L}_{A,C} \cup \mathcal{D}_{A,C}$$

Figure 3 illustrates the definitions of $\mathcal{L}_{A,C}$ and $\mathcal{D}_{A,C}$. When a component is removed from the assembly, all components which do not participate anymore in the assembly's completeness can be removed. Components from the dead connected components set $\mathcal{D}_{A,C}$ can be removed from the assembly because they only participated in the removed component's coherence. Indeed, as dependencies are modelled by edges of the graph, if there are unconnected subgraphs that are not needed to implement the functional objectives (which we call subgraphs of dead components), these subgraphs are useless (no dependency links them to the parts of the graphs that contain functional objectives).

Removing the dead components is a necessary step because keeping useless components add useless dependencies that make the resulting assembly considerably larger, thus complicating the building process, making the validity checks more difficult and making the assembly more subject to failures, less open for extensions, etc. Let us just also note that the components in $\mathcal{D}_{A,C}$ are dead components but that there still might be useless components in $\mathcal{L}_{A,C}$ (those we keep). We are considering future improvements that would exploit the protocols to improve the detection of dead components.

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

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

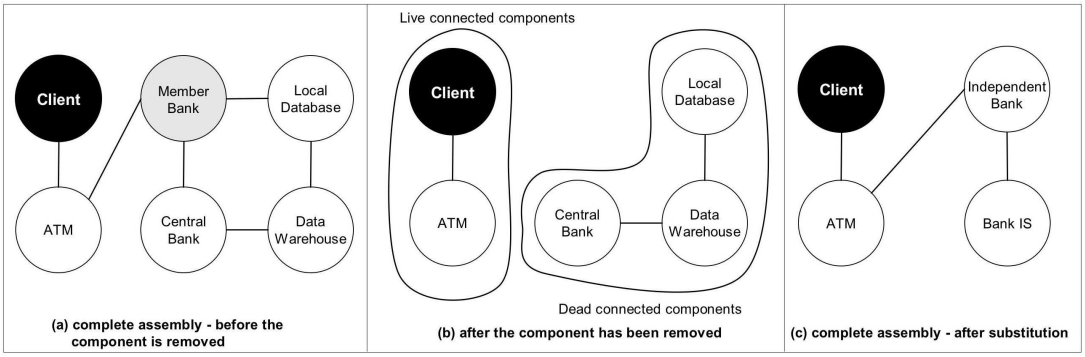


Figure 4. Evolution scenario on the ATM example: removal of the *MemberBank* component

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

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

Evolution Scenario. For our *ATM* example, Figure 4(a) represents the graph corresponding to the example of Figure 2. The *Client* node represents the *Client* component which contains a functional objective. The other nodes (*MemberBank*, *ATM*, *CentralBank*, *LocalDatabase* and *DataWarehouse*) represent components which do not contain any functional objective. We also assume there is a component repository, which will be searched for possible replacement components. Figure 4(b) shows that the partial component assembly from $\mathcal{L}_{ATMexample,MemberBank}$ is not complete because the *ATM* component has become incoherent after the *MemberBank* component and the three now consequently dead components ($\mathcal{D}_{ATMexample,MemberBank} = \{CentralBank, LocalDatabase, DataWarehouse\}$)

have been removed. To complete the assembly, new components must be added. Figure 4(c) sketches the result of this re-building process: The *IndependentBank* component is connected to the *BankIS* component and they both replace the components that had been removed to complete the *ATM* example assembly.

Figure 5 details the resulting architecture. In this example, *MemberBank* is the component to remove. When it is removed, completeness of the architecture is lost. Indeed, the *ATM* component is not locally coherent anymore. Its *Money_Withdraw* composite port is not coherent because the primitive port *Money_Transaction* is not connected while the *Money_Dialogue* primitive port is not shared and still connected. The *CentralBank*, *LocalDatabase* and *DataWarehouse* components constitute the $\mathcal{D}_{ATMexample,MemberBank}$ graph and can also be removed. Completeness is reached by selecting and connecting new components. In this example, an *IndependentBank* component is connected to the *ATM* component through its *Money_Transaction* primitive port. At this step, the assembly is not yet complete because all the components are not yet coherent. Indeed, the *IndependentBank* component is not coherent because its *Manage_Withdraw* composite port is not coherent. Another component is thus added to the assembly: the *BankIS* component is connected to the *IndependentBank* component through its *Request_Data* primitive port. At this point, the assembly is complete. As a result, one can then consider that the removed component has been replaced by an assembly composed of the *IndependentBank* and the *BankIS* components.

Optimization of the Re-building Process using Strategies and Heuristics

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

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

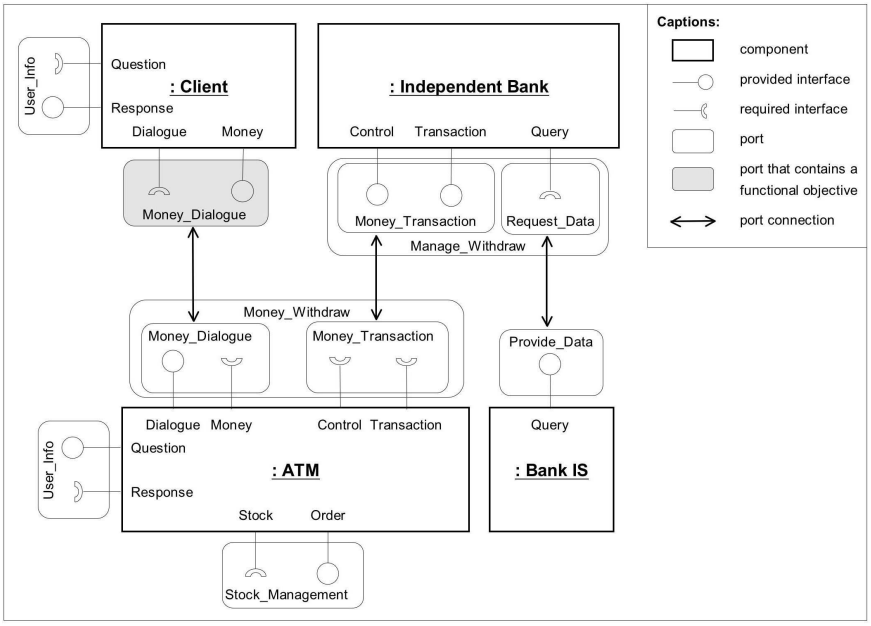


Figure 5. Dynamic reconfiguration of the assembly

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

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

Look-ahead (LA) Strategy. An estimate can be used to predict if traversing the current construction branch can lead to a minimal solution. This estimate is based on the minimum number of primitive port connections required to complete the building process. As soon as the sum of the estimate and the number of already existing connections is larger than the current bound, the branch can be pruned. A simple example of such an estimate is the number

of ports in the FO-set divided by two, which corresponds to a lower bound of the number of connections needed to connect the ports that already are in the FO-set (in the most optimistic case, each port from the FO-set can connect to another port from the FO-set thus adding no new dependency and, moreover, satisfying two dependencies at once). A more selective and realistic estimate consists in calculating how many of primitive port pairs from the FO-set can be connected with one another. The number of remaining connections is higher than the cardinality of the FO-set minus the number of primitive port pairs.

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

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

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

Implementation and Experimentation

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

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

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

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

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

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

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

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

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

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

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

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

Table III. Synthetic view of results for reconfiguration experiments

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

Evaluation of the dynamic reconfiguration approach. Our dynamic reconfiguration approach has been tested in the same environment used to test the building process. The experimentation context is defined by the variable values shown in last column of Table I. We generated 10 component repositories for this context. Then, to test our solution for evolution, we started from a generated complete component assembly from which a randomly chosen component was removed. The substitution process was then triggered by considering that the removed component was not available anymore. We ran the dynamic reconfiguration process 40 times, each time with a newly generated assembly (varying the set of functional objectives) and a new component to remove. Results are synthesized in Table III. Those experiments showed that our solution provides alternative substitution possibilities (compared to existing one-to-one substitution mechanisms), thus is more flexible because it does not depend on the presence of a component that can exactly match (the role of) the removed one. In some situations (20%), no solution exists—the repository does not contain components that can be combined to be substituted to the removed one— but among the solved situations, 81% are solved

thanks our many-to-one substitution proposal (compared to only 19% that can be solved with usual one-to-one substitution techniques). Furthermore, the resulting substitution was usually many-to-one. Also, we noticed that the complexity of the mechanism exposed here is not higher than the complexity of the complete building process—which was efficient thanks to the optimization strategies and heuristics.

Conclusion

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

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

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

REFERENCES

1. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
2. Frantisek Plásil, Dusan Balek, and Radovan Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. of the Int. Conf. on Configurable Distributed Systems*, pages 43–52, Washington, DC, USA, 1998. IEEE Computer Society.
3. E. Bruneton, T. Coupaye, and JB. Stefani. Fractal specification - v 2.0.3, February 2004. `http://fractal.objectweb.org/specification/index.html` [3 July 2008].
4. Bart George, Régis Fleurquin, and Salah Sadou. A substitution model for software components. In *Proc. of the 2006 ECOOP Workshop on Quantitative Approaches on Object-Oriented Software Engineering (QaOOSE'06)*, Nantes, France, July 2006.

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

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

AUTHORS' BIOGRAPHIES

Nicolas Desnos has recently (2008) obtained his PhD from Montpellier 2 University. He worked on defining an autonomic process to build complete software component assemblies. His research interests are component-based software engineering and heuristic search techniques.

Marianne Huchard is a fulltime professor at the LIRMM laboratory (CNRS and Université Montpellier 2) since 2004. Her research interests include Formal Concept Analysis (FCA) and Relational Concept Analysis (RCA) for software engineering (class model reengineering, component directories), model driven engineering for FCA and RCA applications as well as component-based software engineering.

Guy Tremblay is a fulltime professor in the computer science department of the Université du Québec à Montréal (UQAM) since 1985. He is also a member of the LAboratory for research on Technology for E-commercE (LATECE). His research interests include parallel programming, model checking techniques, business process modeling languages and formal methods for (web) service composition and software components.

Christelle Urtado is a fulltime assisant-professor at the Ecole des Mines d'Alès since 1999. Her resarch interests include self-* approaches for component-based software engineering (component self-assembly, component evolution, component directories) and fault tolerance (exception handling) in component-based or agent-based software systems. She also interests in helping designers build and maintain complex software systems.

Sylvain Vauttier is a fulltime assistant-professor at Ecole des Mines d'Alès since 2000. His research interests encompass component and agent-based software engineering techniques. His work, focused on behavior composition mechanisms, is more recently applied on the autonomous construction and evolution of software deployed on ambiant intelligence environments.

Appendix C

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

This journal article [20] has been published in the International Journal of General Systems, in 2009. It presents work realized in collaboration with Gabriela Arévalo from Argentina.

This work states the theoretical basis of an on-the-fly indexing of component directories using Formal Concept Analysis, based on the syntactic description of the components' required and provided services. In such directories, components are more intelligibly organized and new abstract and highly reusable component external descriptions are suggested. But over all, such substitutability-based indexing eases component search which is useful in automating both component assembly and component substitution.

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

Research paper

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

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

Christelle Urtado [b] and Sylvain Vauttier [b]

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

(*released August 2008*)

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

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

1. Introduction

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

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

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

ISSN: 0308-1079 print/ISSN 1563-5104 online

DOI: 10.1080/0308107YYxxxxxxxx
http://www.informaworld.com

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

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

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

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

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

2. Functionality signatures and interface syntactical compatibility

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

2.1 *Functionality signature compatibility in object-oriented programming*

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

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

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

2.2 *Functionality signatures and component interface specification*

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

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

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

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

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

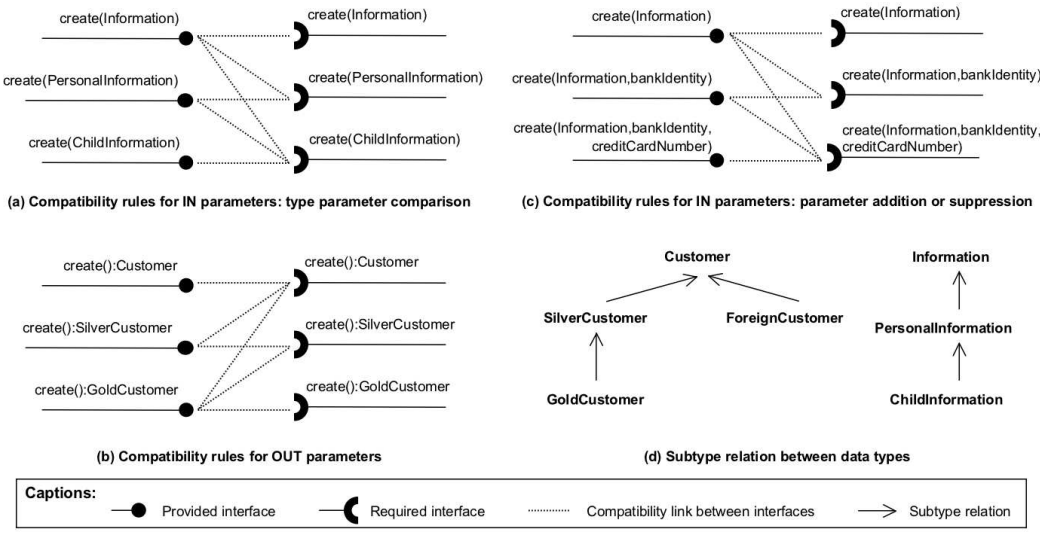


Figure 1. Interface compatibility when types and number of parameters vary.

2.3 *Functionality signature specialization and provided interface substitution*

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

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

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

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

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

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

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

To be true, the predicate entails that:

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

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

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

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

2.4 *Functionality signature specialization and required interface substitution*

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

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

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

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

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

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

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

To be true, the predicate entails that:

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

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

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

2.5 *Functionality signature specialization and interface connection*

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

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

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

To be true, the predicate entails that:

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

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

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

2.6 *Functionality signature specialization and parameter addition or suppression*

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

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

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

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

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

2.7 *Discussion*

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

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

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

3. Lattice of functionality signatures

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

3.1 *A survival kit for Formal Concept Analysis*

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

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

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

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

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

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

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

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

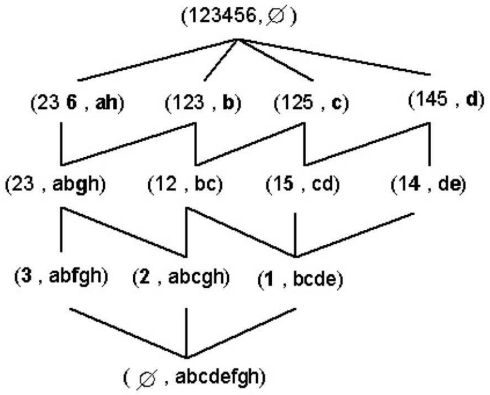


Figure 2. Hasse diagram of the concept lattice $\mathcal{L}$.

3.2 *Example of an online bookstore application*

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

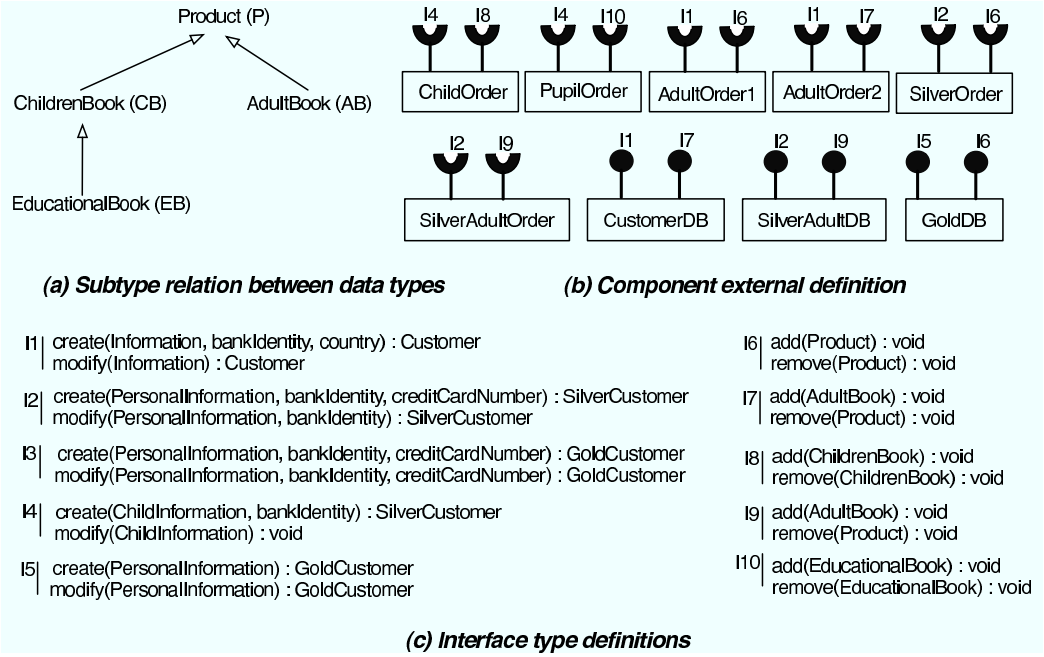


Figure 3. Data types, interfaces and components of an online bookstore application.

3.3 *Building the functionality signature lattice*

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

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

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

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

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

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

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

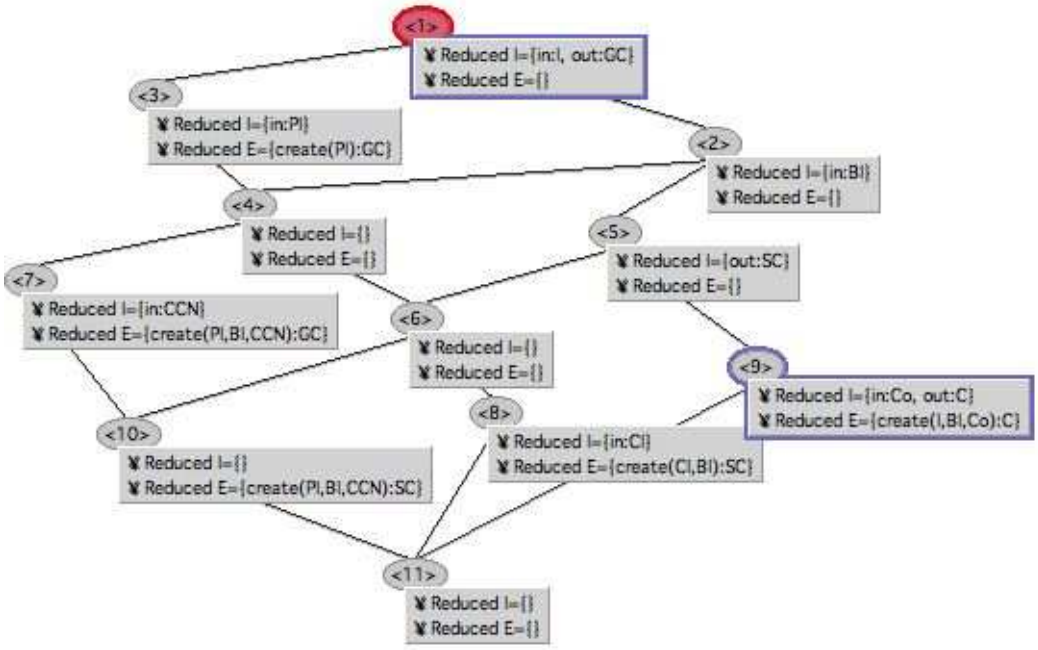


Figure 4. Signature lattice $\mathcal{L}_{create}$ for the `create` functionalities.

3.4 *Using the functionality signature lattice*

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

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

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

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

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

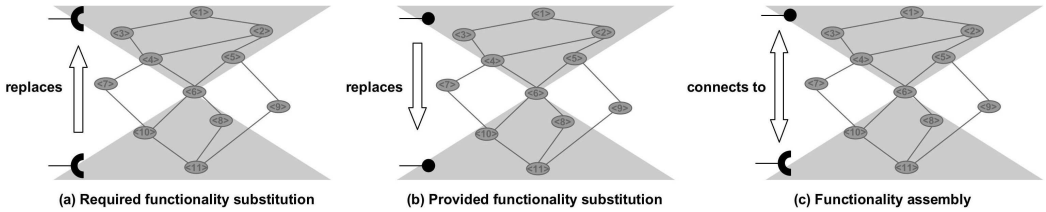


Figure 5. Interpretation of the lattice of functionality signatures.

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

4. Interface lattice

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

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

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

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

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

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

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

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

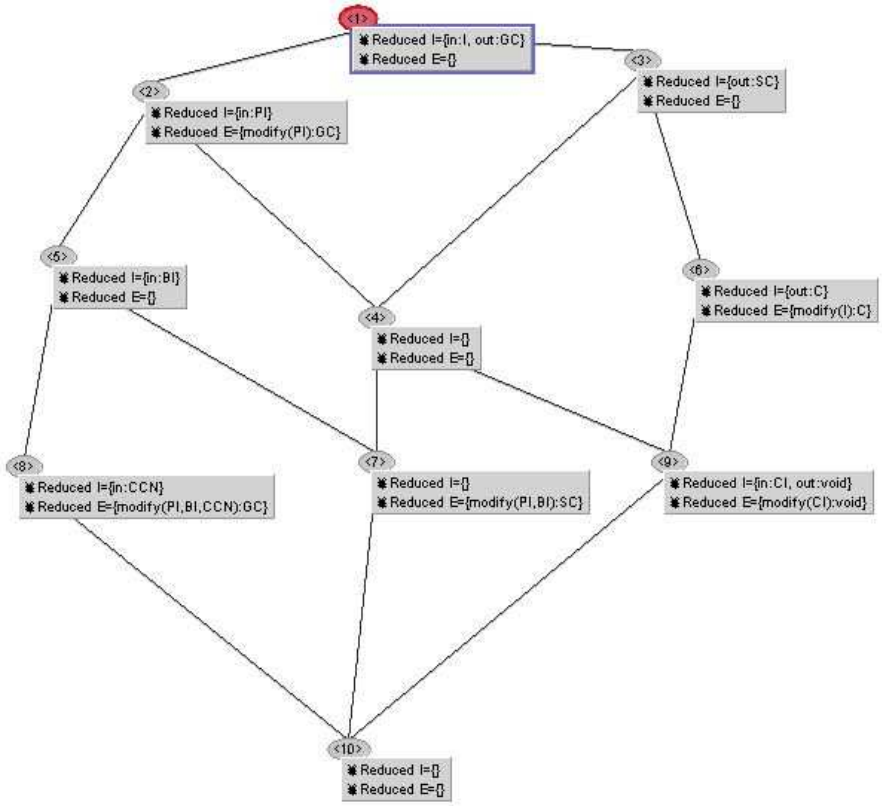


Figure 6. Signature lattice $\mathcal{L}_{modify}$ for the `modify` functionalities

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

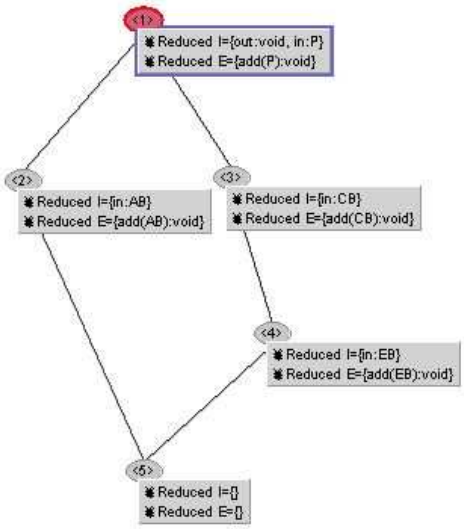


Figure 7. Signature lattice $\mathcal{L}_{add}$ for the `add` functionalities. The lattice $\mathcal{L}_{remove}$, isomorphic to $\mathcal{L}_{add}$, is not represented.

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

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

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

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

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

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

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

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

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

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

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

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

just a flat set of artifacts.

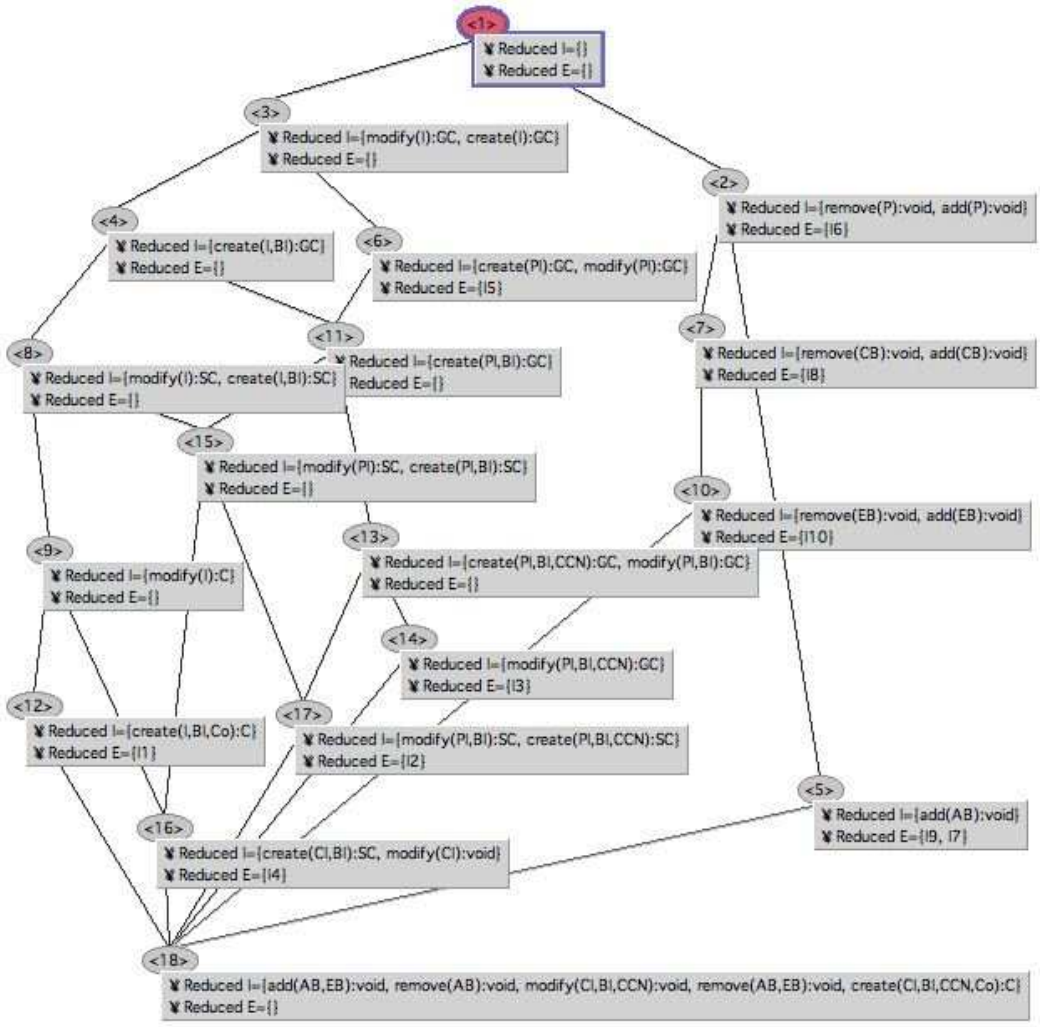


Figure 8. Interface lattice $\mathcal{L}_I$ using the functionality signature lattice.

5. Lattice of component types

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

5.1 *Definition of the lattice of component types*

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

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

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

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

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

Table 6. New canonical interfaces.

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

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

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

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

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

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

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

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

5.2 *Usage of the lattice of component types*

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

5.2.1 Emergence of new component types

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

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

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

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

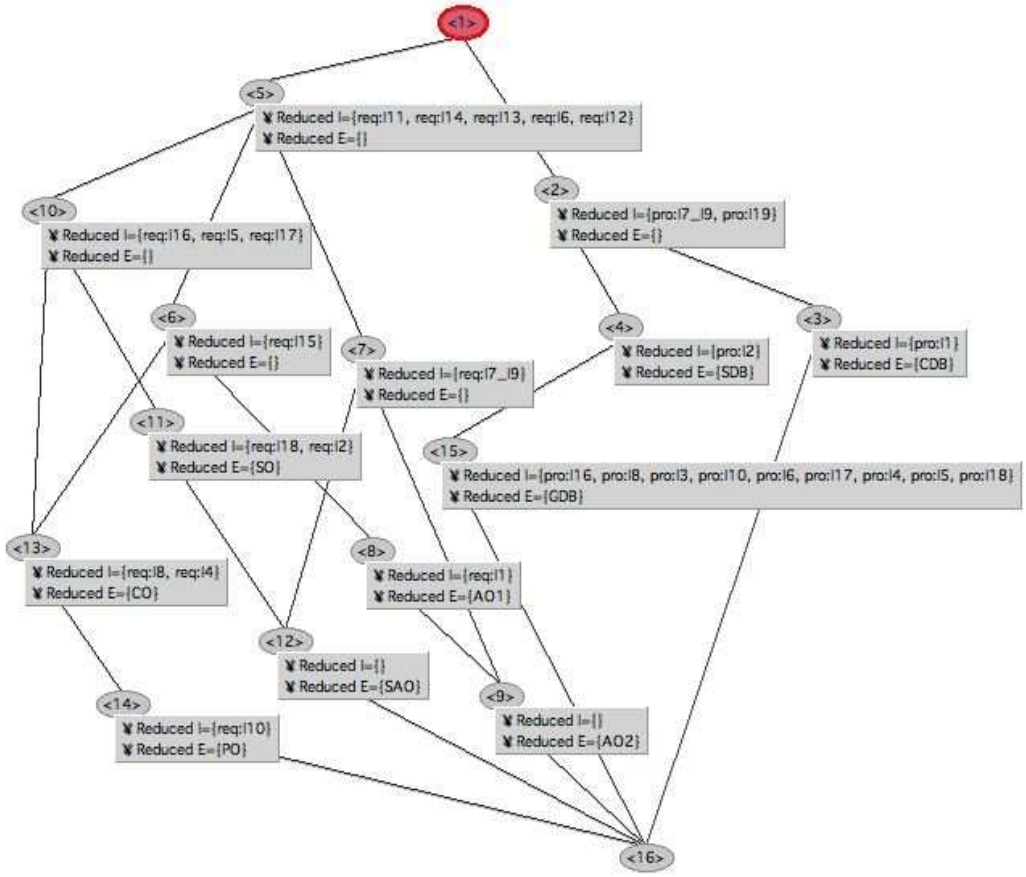


Figure 9. Lattice $\mathcal{L}_C$ of component types using the interface lattice.

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

5.2.2 Substitution and connection

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

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

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

5.2.3 Reingeneering and building generic architectures

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

5.2.4 Architecture abstraction

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

6. Related work

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

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

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

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

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

7. Conclusion and future work directions

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

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

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

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

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

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

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

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix D

Architecture-centric component-based development needs a three-level ADL

This conference paper [117] has been presented at the 4th European Conference on Software Architecture (ECSA 2010). It presents part of the work realized during the PhD of Huaxi (Yulin) Zhang.

It introduces the Dedal ADL, with its three architecture representation levels tailored for reused-centered architecture development in three steps.

Architecture specifications first capture abstract and ideal architectures imagined by architects to meet requirements. Specifications do not describe complete component types but only component roles (usages).

Architecture configurations then capture implementation decisions, as the architects select specific component classes from the repository to implement component roles.

Finally, architecture assemblies define how components instances are created and initialized to customize the deployment of architectures in their own execution contexts.

The refinement relationships between these architecture representations are also discussed.

Architecture-centric component-based development needs a three-level ADL

Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier

LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Huaxi.Zhang, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

Abstract. Architecture-centric, component-based development intensively reuses components from repositories. Such development processes produce architecture definitions, using architecture description languages (ADLs). This paper proposes a three step process. Architecture specifications first capture abstract and ideal architectures imagined by architects to meet requirements. Specifications do not describe complete component types but only component roles (usages). Architecture configurations then capture implementation decisions, as the architects select specific component classes from the repository to implement component roles. Finally, architecture assemblies define how components instances are created and initialized to customize the deployment of architectures in their own execution contexts. This development process is supported by a three-level ADL which enables the separate definition of these three representations. The refinement relationships between these architecture representations are also discussed.

1 Introduction

Component-based software development (CBSD) consists in two activities: the development of software components for reuse and the development of software applications by the reuse of components. The first activity can be managed by classical software development processes, with an analysis, a design and then a coding phase. The produced software modules, encapsulated as component classes, are then stored and indexed in repositories to be reused later on. The second activity corresponds to a more specific and still scarcely studied development processes. We propose an architecture-centric development process that aims at defining the structure of an application as a set of reused components and a set of connections between them, using a dedicated language called an Architecture Description Language (ADL). This process is structured in three steps, through which architecture definitions are gradually refined, from abstract to concrete representations. After a classical analysis step, architecture specifications first capture design decisions as ideal architectures imagined by architects to meet the requirements. Specifications do not describe complete component types but only component roles (usages). These roles are used to search for matching component classes in repositories. Specification and roles are thus key concepts to integrate component reuse effectively in the development process.

Second, architecture configurations capture implementation decisions, as the architects select specific component classes to implement component roles. Finally, architecture assemblies define how components instances are created and initialized to customize the deployment of architectures in different execution contexts. Our process is supported by an three-leveled dedicated ADL, called Dedal, which enables the explicit and separate definitions of architecture specifications, configurations and assemblies. This way, a single abstract architecture definition can be refined into many concrete architecture definitions, to foster not only the reuse of components but also of architectures. The refinement relationships between these separate architecture representations — i.e. the relationship between the component roles, classes and instances they are composed of — are proposed to control and verify the global coherence of these multi-level architecture definitions.

The remaining of this paper is organized as follows. Section 2 introduces our proposed architecture-centric, reuse-based development process. It studies how existing ADLs are suitable for it. Section 3 presents the different component description levels supported in Dedal, our proposed ADL to support this development process. Section 4 presents the different architecture description levels which can be expressed in Dedal, along with the refinement relations between them. Section 6 concludes with future work directions.

2 Software Architectures in CBD

2.1 A Development Process for Component Reuse

Component-based software development is characterized by its implementation of the "reuse in the large" principle. Reusing existing (off-the-shelf) software components therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [1, 2]. Figure 1 illustrates our vision of such a development process which is classically divided in two:

- the component development process (sometimes referred to as component development *for* reuse), which is not detailed here. This development process is the producer of components that are stored in repositories for later consumption by the component reuse process.
- and, the component-based software development process (referred to as component-based software development *by* reuse) that describes how previously developed software components can be used for software development (and how this reuse process impacts the way software is built).

The proposed component-based software development process deliberately focuses on the produced artifacts (architecture descriptions, as models of the software) for each development step. For simplicity's sake, it is also exclusively "reuse-centered" and does not describe how components should be developed

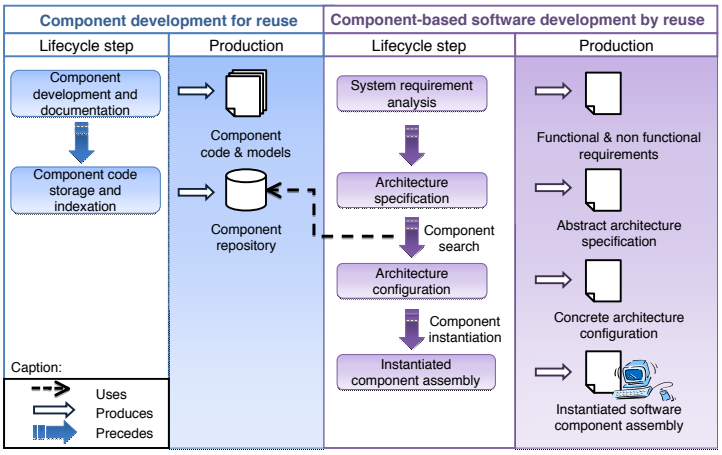


Fig. 1. Component-based software development process

from scratch if no component is found that matches or closely matches specifications, adapted if no existing component type perfectly matches specifications, tested and integrated or, physically deployed.

In this component-based software development process, software is considered to be produced by the reuse of components that have previously been stored and indexed in a component repository. It decomposes in three steps each of which produces a description that models the view of the architecture at this development step:

1. *Model of requirements.* After a classical requirement analysis step, architects establish the abstract architecture specification. They define which functionalities should be supplied by components, which interfaces should be exported by components, and how interfaces should connect to build a software system that meets the requirements.
2. *Model of design.* In a second step, architects create architecture configurations that define the sets of component implementations (classes) by searching and selecting from the component repository. Abstract component types from the architecture specification then become concrete component types in architecture configurations.
3. *Model of runtime.* In a third step, configurations are instantiated into component instance assemblies and deployed to executable software applications.

The claim of this paper is that an architectural description should correspond to each of the three steps of the component-based software development process. In other words, architectures should be described from all specification (model of requirements), configuration (model of design) and assembly (model of runtime) point of views. These three descriptions should reflect the architect's design decisions at each step of the development cycle and be expressed using an adequate

ADL. State-of-the-art ADLs have been analyzed from this perspective, trying to answer the following questions (that provide a taxonomy for comparison):

- Do existing ADLs support multiple view representations?
- If so, are these views used to reflect successive development steps?
- In cases where several descriptions of a given architecture coexist, which development step can they be associated to?
- Which information on software is captured? In which view / level representation?

2.2 Expressiveness of Existing ADLs

A software system architecture [3] gathers design decisions on the system. It is expressed using an ADL which, in most cases, provides information on the structure of the software system listing the components and connectors the system is composed of. Quality attributes are sometimes provided (*e.g.* xADL [4]). The dynamic behavior of systems is often described (*e.g.* C2SADEL [5], Wright [6], SOFA [7]) but their descriptions are not homogeneous as various technologies (*e.g.* message-based communication, CSPs, regular expressions) are used.

When systems are too complex to easily be described, two classical mechanisms can be used to split descriptions into smaller ones. *Hierarchical decomposition* enables to view the system at various granularities (*e.g.* Darwin [8], SOFA [7] or Fractal ADL [9]). Systems are composed of sub-systems that can further be described at a finer level. *Thematic decomposition* amounts to consider the system from distinct viewpoints (*e.g.* syntactic and behavioral diagrams of UML [10]). Whole systems are seen from several partial viewpoints that make each description focus on some system attributes.

Systems can also be described at various steps of their life-cycles. To our knowledge, no ADL really includes this "time" dimension. Some works such as UML [10] or Taylor *et al.* [3] implement or describe close notions. UML makes it possible to describe object-oriented software at various life-cycle steps but this capability is not transposed in their component model. Taylor *et al.* [3] distinguish two description levels for architectures at design and programming time, respectively called perspective (or as-intended) and descriptive (or as-realized) architectures. However, as far as we know, they do not propose any ADL or metamodel to concretely implement these two architecture descriptions. Garlan *et al.* [11] propose a three-layer framework (task, model and runtime layers) and points out the importance of three levels for dynamic software evolution management. Beside their having close notions, these existing works do not propose such descriptions that would follow the three identified steps of component-based software development.

We then examine the representative ADLs to see which levels of architecture descriptions are supported (as shown in Tables 1 and 2). As far as we know, the studied ADLs unfortunately do not enable the three levels that correspond to lifecycle steps to be all described. This analysis results in requirements for the language presented in this paper:

Table 1. Expressiveness of existing ADLs — Modeling of the three lifecycle steps

ADL	Specification	Configuration	Assembly
C2SADEL	✓	✓	×
Wright	×	✓	×
Darwin	×	✓	×
Unicon	×	✓	×
SOFA 2.0	×	✓	×
FractalADL	×	✓	×
xADL 2.0	×	✓	✓

Table 2. Expressiveness of existing ADLs — Component representations

ADL	Abstract component type	Concrete component type	Component class	Component instance
C2SADEL	×	✓	✓	×
Wright	×	✓	✓	×
Darwin	×	✓	✓	×
Unicon	×	✓	✓	×
SOFA 2.0	×	✓	✓	×
FractalADL	×	✓	✓	×
xADL 2.0	×	✓	✓	✓

1. No ADLs presented in Table 1 is tailored to CBD. Switching to such a reuse-centered development process shall impact the description language.
2. No ADLs presented in Table 1 models component types in an abstract way in order to support the search and selection of concrete component in component repositories. Concrete components in architecture configurations should not be strictly identical to abstract component types described in their architecture specification. As components pre-exist, the specification should define abstract (ideal) and partial component types while configurations describe concrete (satisfying) components that are going to be used (as claimed by Taylor *et al.* [3]).
3. Connectors should not necessary be explicit but the architect should have the possibility to explicit them when needed. Explicit connectors model specific connection types and can be reused from one design to another. However, in most situations, connectors can be system-generated and thus remain implicit for simplicity's sake.
4. Most ADLs do not model the running system (assembly level) or component instances, except xADL 2.0. ADLs should include some description on how components classes are instantiated and what are the characteristics of the running assemblies (constraints on component state values).
5. Components should possibly be primitive (implemented by an implementation class) or hierarchically composed of components (implemented by a configuration).

6. Component types should be reusable. This implies that their description is modularized (outside architectures).
7. Both structural and behavioral viewpoints should be provided for both components and architectures.

2.3 Example of a Bicycle Rental System

Figure 2 shows the example used throughout the paper: the architecture specification of a bicycle rental system (BRS). A *BikerGUI* component manages a user interface. It cooperates with a *Session* component which handles user commands. The *Session* component cooperates with the *Account* and *Bike&Course* components to identify the user, check the balance of its account, assign him an available bike and then calculate the price of the trip when the rented bike is returned. In the following sections, we will use a part of this system to illustrate our concepts and ADL syntax.

The two following sections present Dedal, the proposed ADL which spans the three levels of architecture descriptions. Dedal enables the description of abstract architecture specifications, concrete architecture configurations and instantiated component assemblies. It also supports a controlled architecture evolution process the description of which is out of the scope of this paper (see [12] for this aspect).

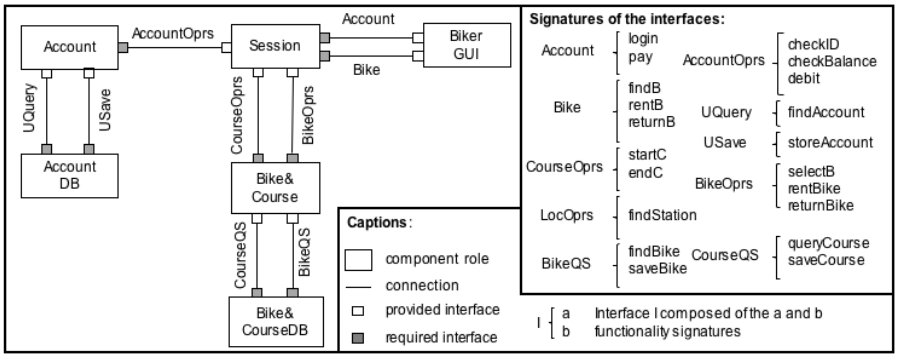


Fig. 2. BRS abstract architecture specification

3 Component Descriptions in the Three Levels of Dedal

Dedal models architectures at three separate abstraction levels, each of which contains different forms of components and connectors. For now, Dedal mainly focuses on modeling components. At the specification level, components are modeled as roles which are requirement models for concrete component search. These specifications thus are abstract and partial. At the configuration level, components are modeled as (whole) component classes which realize the specifications.

Several component classes might correspond to a single component role as there might exist several concrete realizations of a single specification. At the assembly level, concrete component classes are instantiated into component instances that represent runtime components and their parameterizations. Figure 3 shows a complete example of components at three levels.

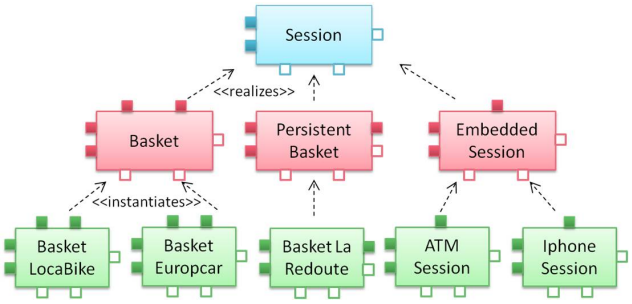


Fig. 3. The *Session* component role, some possible concrete realizations and some of their instantiations

3.1 Components in Abstract Architecture Specifications

Component roles model abstract component types in that they describe the roles components should play in the system. A component role lists the minimum list of interfaces (both required and provided) the component should expose and the component behavior protocol that describes the behavior of the component in the architecture (dynamics of the architecture). As they define the requirements of the architect (its ideal view) to guide the search for corresponding concrete components, component roles are abstract and partial component representations (*e.g. Session* component role on Fig. 3). Dedal uses the protocol syntax of SOFA [7] to describe component behavior as regular expressions[1]. Other formalisms could have been used instead; the notation chosen is interesting as it is compact and is implemented as an extension of the Fractal component model we used for or experimentation, with companion verification tools. Component protocols capture the behavior of components in their context describing all valid sequences of emitted function calls (emitted by the component and addressed to neighbor components) and received function calls (received by the component from neighbor components). As component roles are abstract component specifications, Dedal modularly describes them outside architecture specifications, so as they can be reused from a specification to another (which would not be possible if they were embedded). Figure 4 shows the description of the *Session* component role. This description contains (a part of) the SOFA-like description of its behavior.

[1] `!i.m` (*resp.* `?i.m`) denotes an outgoing (*resp.* incoming) call of method `m` on interface `i`. `A+B` is for `A` or `B` (exclusive or) and `A;B` for `B` after `A` (sequence).

```
component_role Session
 required_interfaces BikeOprs; CourseOprs; AccountOprs
 provided_interfaces Account; Bike
 component_behavior
  (!Session.Bike.findB,
  ?Session.BikeOprs.findB;)
  +
  (!Session.Account.login,
  ?Session.AccountOprs.checkID;)
  . . .
```

Fig. 4. *Session* component role

3.2 Components in Concrete Architecture Configurations

At configuration level, components are modeled in two ways with *component types* and *component classes*. Figure 5 provides a close-up view of the relationships between a component role (that model an abstract and partial view of a required component), a component type that models the complete type of some existing concrete implementation, a component class that represent the concrete component implementation and a parameterized component instance.

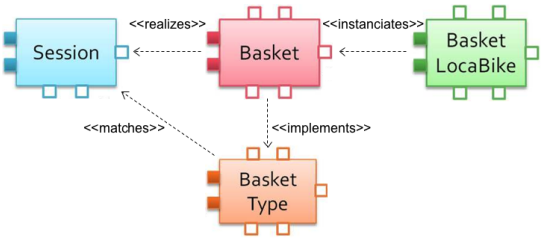


Fig. 5. *BikeCourseDBClass* composite component

Component types represent the full types of at least one (maybe several) existing component implementations. They are defined by describing the interfaces and behavior of these component classes. Component types are reusable as they can be implemented by multiple component classes which possess the same interfaces and component behavior. The *BasketType* component type description of Fig. 6 is an example of component type description.

Component classes represent concrete component implementations. Each component class points to the component type it implements. Component classes can either be primitive or composite.

Primitive component classes (*e.g. Basket* as described in Fig. 7) define the reused components by describing their interfaces, behavior, version[2] and imple-

[2] This information (as well as all the versioning information included in other descriptions later on) serves evolution management purposes that are not described in this paper. For more information, the interested reader might refer to [12].

```
component_type BasketType
  required_interfaces BikeOprs; CourseOprs; AccountOprs; CampusOprs;
AccessoryOprs
  provided_interfaces Account; Bike
  component_behavior
    (!BasketType.Bike.findB,
    ?BasketType.BikeOprs.findB;)
    +
    (!BasketType.Account.login,
    ?BasketType.AccountOprs.checkID;)
    . . .
```

Fig. 6. Description of the *BasketType* component type

menting class. Existing models usually do not include links to the implementaing class as they assume there is a single implementation. In Dedal, components can thus have several implementations (which can be useful to have implementations versioned in such cases as software product lines management).

```
component_class Basket
 implements BasketType
 using fr.ema.locaBike.Basket
 attributes string company; string currency
```

Fig. 7. The *Basket* (primitive) component class description

Composite component classes will be introduced in Sect.4.2. Both primitive and composite component classes can export an attribute list (as exemplified on Fig. 7 and 11). Attributes are not mandatory but can be declared as observable / visible properties for component classes so as to be able to set assembly constraints on attribute values in the instantiated component assembly level.

3.3 Components in Instantiated Software Component Assemblies

Component instances document the real artifacts that are connected together in an assembly at runtime. They are instantiated from the corresponding component classes. They might define constraints on components' attributes that reflect design decisions impacting component states (attribute values) over time. They also set the initial component state by initializing attributes values.

```
component_instance BasketLocaBike
  instance_of Basket (1.0)
  initiation_state company="LocaBikecurrency"; currency=="Euro."
```

Fig. 8. The *BasketLocaBike* component instance description

4 Three Levels of Architecture Description in Dedal

4.1 Abstract Architecture Specifications

Abstract architecture specifications (AASs) are the first level of software architecture descriptions. They provide a generic definition of the global structure and behavior of software systems according to previously identified functional requirements. They model the requirements expressed by the architect to serve as a basis to search for concrete component to create concrete architecture configurations. These architecture specifications are abstract and partial: they do not identify concrete component types that are going to be instantiated in the software system. They only describe the "ideal" component types from the application point of view. Types of concrete components need not be identical to abstract types. As CBD processes favors component reuse, component type compatibility should be more permissive than strict identity but still guarantee safety of use. Compatible concrete component types can, for example, provide more functionalities than strictly specified (extra functionality will remain unused) or provide more generic functionalities (use of polymorphism of object-oriented languages)[3].

```
specification BRSSpec
 component_roles
  BikeCourse; BikeCourseDB
  ...
 connections
  connection connection1
   client BikeCourse.BikeQS
   server BikeCourseDB.BikeQS
  connection connection2
   client BikeCourse.CourseQS
   server BikeCourseDB.CourseQS
  ...
 architecture_behavior
  (!BikeCourse.BikeOprs.selectBike;
  ?BikeCourse.BikeQS.findBike;
  !BikeCourseDB.BikeQS.findBike;)
  +
  (!BikeCourse.CourseOprs.startC;
  ?BikeCourse.CourseQS.findCourse;
  !BikeCourseDB.CourseQS.saveCourse;)
  ...
 version 1.0
```

Fig. 9. AAS of the BRS (partial)

In Dedal, an AAS is composed of a set of component roles, a set of connections and its architecture behavior. Figure 9 provides an example of the AAS for the BRS. For readability reasons, this description represents only a small

[3] The reader further interested about component compatibility can refer to authors' work on component repositories [13] and component substitution [14].

part of the BRS AAS depicted in Fig. 2. **Connections** make interactions between two components possible. They define which component interfaces are bound together. *connection1* and *connection2* from Fig. 9 are such connections. **Architecture behaviors** describe the protocols of complete architectures – meaning all possible interactions between their constituent components. As for component protocols, the syntax used is that of SOFA protocols[4]. Compatibility between individual component protocols and the protocol of their containing architecture as well as compatibility between the protocols of two connected components is not studied in this work as we interface our language with corresponding mechanisms (trace inclusion) from SOFA. Figure 9, that describes the BRS architecture specification, contains the BRS architecture protocol. The reader can intuitively check that the protocol of the *BikeCourse* component role is compatible with ("included" in) the protocol of the BRS architecture.

4.2 Concrete Architecture Configurations

Concrete architecture configurations (CACs) are the second level of system architecture descriptions. They result from the search and selection of real component types and classes in a component repository. These component types must match abstract component descriptions from the architecture but need not be identical; compatibility is sufficient. Component classes must be valid implementations of their declared component type. CACs describe the architecture from an implementation viewpoint (by assigning component roles to existing component types). Architecture configurations thus list the concrete component and

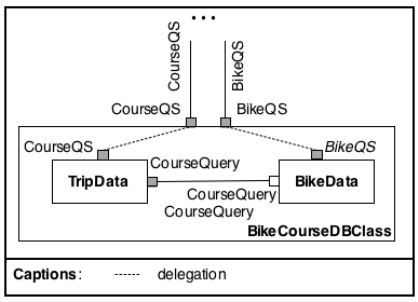
```
configuration BRSConfig
 implements BRSSpec (1.0)
 component_classes
  BikeTrip (1.0) as BikeCourse;
  BikeCourseDBClass (1.0) as BikeCourseDB
 version 1.0
```

Fig. 10. A possible CAC for the BRS

connector classes which compose a specific version of a software application. The architecture of a given software is thus defined by a unique AAS and possibly several CACs. For a given software, each architecture configuration must conform to the architecture specification. This means that each component or connector class used in an architecture configuration must be a legal implementation of the corresponding component role or connection in the architecture specification. Figure 10 describes the architecture configuration of the BRS. The explicit description of **connector classes** is possible (as exemplified on Fig. 12) but not mandatory. In cases where they are implicit, we consider connectors as

[4] `!c.i.m` (*resp.* `?c.i.m`) denotes an outgoing (*resp.* incoming) call of method `m` on interface `i` of component `c`.

generic entities which are provided by containers (execution environments) in which configurations are deployed. Connections are automatically administered by containers at runtime to manage the instantiation of configurations. In cases where connectors are explicitly added, their descriptions define the specific connector classes that reflect design choices and that must be used to manage special communication, coordination, and mediation schemes. **Composite component classes** are components the implementation of which is not a simple class but a complete configuration that differ from the above described configurations in that it has some unconnected interfaces. The composite component class concept enables hierarchical composition of architectures which has been identified as an effective means to manage system complexity and concretely implement reuse (as whole configurations can be considered as coarser grained components). Composite component classes further define how unconnected interfaces from the inner configuration can be delegated to interfaces of the composite component. As for provided and required interfaces in primitive components, delegated interfaces are implementations of the corresponding provided and required interfaces in the corresponding component role. Figures 11 and 12 give the example of the composite component class *BikeCourseDBClass* that implements the *BikeCourseDB* role where the *BikeQS* provided interface of the *BikeData* component inside the *BikeCourseDBConfig* configuration is delegated as a provided interface of the composite component that implements the *BikeQS* interface of the *BikeCourseDB* component role. Figure 11 shows a graphical representation of the same *BikeCourseDBClass* component.




```
component_class BikeCourseDBClass
 implements BikeCourseDB
 using BikeCourseDBConfig (1.0)
 delegated_interfaces
  provided
   BikeCourseDBConfig.BikeData.BikeQS
   as BikeCourseDB.BikeQS
  provided
   BikeCourseDBConfig.TripData.CourseQS
   as BikeCourseDB.CourseQS
 version 1.0
 attributes company
```

Fig. 11. The *BikeCourseDBClass* composite component class and its description

Conformance between an AAS and a CAC is a matter of conformance between component roles and the component classes that supposedly implement them. Many conformance relations could be defined, from stricter to very loose ones. On the one hand, we defend that reused components need not be exactly identical to specifications because being too strict in this matter might seriously decrease the number of reuse opportunities. On the other hand, it is expected from a conformance relation that it enables verifications that guarantees good

```
specification BikeCourseDBSpec
 component_roles
  BikeDB; CourseDB
 connections
  connection ConnectionCourseQuery;
   client BikeDB.CourseQuery
   server CourseDB.CourseQuery
 version 1.0
```

```
configuration BikeCourseDBConfig
 implements BikeCourseDBSpec (1.0)
 component_classes
  BikeData (1.0) as BikeDB;
  TripData (1.0) as CourseDB
 connector_classes
  CourseQuery (1.0) as
ConnectionCourseQuery;
 version 1.0
```

Fig. 12. Descriptions of the *BikeCourseDBSpec* abstract specification and of the *BikeCourseDBConfig* inner configuration

chances that the thought component combination will execute. The rule of the thumb that can be used is that concrete components must provide at least what is the specification declare it provides and require less than what the specification already requires. This translates into rules for interfaces and rules for behavior protocols:

- the provided interfaces list of the concrete component class must contain all the interfaces specified in the component role,
- all the required interfaces of the concrete component class must be specified in the component role,
- the behavior of a component class includes (in the sense of trace inclusions) the behavior specified in the component role.

Variations on these rules can further consider interface specialization rules as in [13]. Figure 7 shows an example of a concrete component class (*BikeTrip*) that has a required interface (*LocOprs*) that is not in the specification (*BikeCourse* component role) it conforms to. In the case of composite components, delegated interfaces of provided (*resp.* required) direction are considered exactly as if they where provided (*resp.* required) interface of primitive components. Indeed, when considered externally, composite components can be seen as if they where primitive. Figure 7 provides an example of the *BikeCourseDBClass* composite component class, that conforms to the specification of the *BikeCourseDB* component role.

4.3 Instantiated Software Component Assemblies

Instantiated software component assemblies (ISCAs) are the third level of system architecture descriptions. They result from the instantiation of the component classes from a configuration. They provide a description of runtime software systems and gather information on their internal states. Indeed, this description level enables the record of state-dependent design decisions [15]. ISCAs list the component and connector instances that compose a runtime software system, the attributes of this software system, and the assembly constraints the component instances are constrained by. Figure 13 gives the description of a software assembly that instantiates the BRS architecture configuration of Fig. 10.

```
assembly BRSAss
 instance_of BRSConfig (1.0)
 component_instances
  BikeTripC1; BikeCourseDBClassC1
 assembly_constraints
  BikeTripC1.currency="Euro.";
  BikeCourseDBClassC1.company=
   BikeTripC1.company
 version 1.0
  component_instance BikeTripC1
   instance_of BikeTrip (1.0)
  component_instance BikeCourseDBClassC1
   instance_of BikeCourseDBClass (1.0)
```

Fig. 13. Component assembly description of the BRS

The explicit description of **connector instances** is possible but not mandatory. In cases where they are implicit, we consider them as generic entities which are provided by containers (execution environments) in which configurations are deployed. In cases where connector instances are explicitly added, their descriptions define the specific attributes that reflect implementation choice to meet different situation. By default, component classes can be instantiated into multiple component instances. When more precise **cardinality** information is needed, it is expressed in component role descriptions using *minInstances* and *maxInstances* that define the minimum and maximum numbers of component instances that are permitted to instantiate from the component class which implements this component role. By this means, component classes do not include this configuration-dependent information and remain reusable. In the assembly level, assembly constraints that restrain the valid number of instances will be checked against the cardinality information defined in the component role (in the specification level). There is no rule to constrain the name of component instances of a given component class. **Assembly constraints** define conditions that must be verified by attributes of some component instances of the assembly, to enforce its consistency. Such assembly constraints are not mandatory. For now, Dedal only permits to list several constraints that must all be enforced and that either:

- limit the possible values for an attribute to a given constant,
- restrain the cardinality of some connection end (*i.e.,*the number of instances of the component class that stands at the end of the connection in the configuration) to a given constant,
- or, enforce equality of the values of two distinct attributes that pertain to two distinct component instances of a given component assembly.

Such assembly constrains are illustrated on Fig. 13 where the value of the *currency* attribute of component *BikeTripC1* is fixed to *Euro* and where the value of the attribute *company* of the *BikeCourseClassDBC1* component must be maintained identical to the value of attribute *company* of component *BikeTripC1.* Another example that involves cardinalities would be expressed as the assembly

constraint *InstanceNbr(BikeTrip)=2* that mean that exactly two component instances of the *BikeTrip* component class should be instantiated in this system. The cardinality of the *BikeTrip* component class is recorded in the *BikeCourse* component role specification. These constraints are very simple and do not yet enable the expression of alternatives, negation, nor the resolution of possible conflicts. Such extended assembly constraint management is one of the perspectives for this work for which we plan to take inspiration from systems that manage architectural styles as constraints sets [6, 16].

Conformance between a CAC and an ISCA is quite straightforward. All component instances of the assembly must be an instance of a corresponding component class from its source configuration (and reciprocally). Conformance also includes the verification that attribute names used in an assembly constraint of some component assembly pertain to the component classes the components of the assembly are instances of. For example, the assembly constraint *BikeTripC1.currency="Euro."* of Fig. 13 has the conformance process check whether the *BikeTrip* component class (from which *BikeTripC1* is instantiated) possesses a *currency* attribute.

5 Implementation of Dedal in the Arch3D tool suite

The Dedal ADL presented in this paper has been implemented in the Arch3D tool suite. The language has been implemented twice: as an XML-based ADL and as a Java-based ADL. The tools also propose a component model which enables to instantiate and manipulate corresponding assemblies at runtime which is extended as an extension of Julia, the open-source java implementation of the Fractal component platform[5]. Our extension of the Fractal platform tools has two purposes: to support the explicit and separate representation of specifications and configurations and, to embed these representations in the component model. The three architecture representations are then available and manipulable at runtime, also providing a full support for evolution management. The *Arch3D Editor* tool provides a graphical console to create, view and modify Dedal-based Fractal architectures. Architects can simultaneously display the different representations of an architecture and work on them.

6 Conclusion

Dedal enables the explicit and separate representations of architecture specifications, configurations and assemblies. Architecture design decisions can thus be precisely captured and traced throughout the development process. The three-level syntax of Dedal supports the expression of requirements by the means of abstract and partial component roles that are used as the main conceptual support for the search of reusable components to be included in configurations. The model of the runtime system (the instanciated component assembly) is rich

[5] `http://fractal.ow2.org/`

enough to serve as the baseis of a full evolution process [12]. A perspective for this work is to experiment the use of Dedal to manage component-based software product lines.

References

1. Crnkovic, I., Chaudron, M., Larsson, S.: Component-based development process and component lifecycle. In: Proc. of the Intl. Conf. on Software Engineering Advances, Papeete, French Polynesia (Oct. 2006) 44
2. Chaudron, M., Crnkovic, I.: Component-based Software Engineering. In: Software Engineering; Principles and Practice. Wiley (2008) 605–628
3. Taylor, R., Medvidovic, N., Dashofy, E.: Software Architecture: Foundations, Theory, and Practice. Wiley (Jan. 2009)
4. Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, XML-based architecture description language. In: Proc. of 2nd WICSA Conf., Amsterdam, The Netherlands (2001) 103–112
5. Medvidovic, N., Rosenblum, D., Taylor, R.: A language and environment for architecture-based software development and evolution. In: Proc. of ICSE Conf., Los Angeles, USA (May 1999) 44–53
6. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. **6**(3) (1997) 213–249
7. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11) (2002) 1056–1076
8. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes **21**(6) (1996) 3–14
9. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. Exper. **36**(11-12) (2006) 1257–1284
10. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, (The 2nd Edition). Addison-Wesley (2005)
11. Garlan, D., Schmerl, B., Chang, J.: Using gauges for architecture-based monitoring and adaptation. In: Proc. of Working Conf. on Complex and Dynamic Systems Architecture, Brisbane, Australia (Dec. 2001)
12. Zhang, H.Y., Urtado, C., Vauttier, S.: Architecture-centric development and evolution processes for component-based software. In: Proc. of 22^{nd} SEKE Conf., Redwood City, USA (July 2010)
13. Aboud, N.A., Arévalo, G., Falleri, J.R., Huchard, M., Tibermacine, C., Urtado, C., Vauttier, S.: Automated architectural component classification using concept lattices. In: Proc. of the Joint WICSA / ECSA Conf., Cambridge, UK (Sept. 2009)
14. Desnos, N., Huchard, M., Tremblay, G., Urtado, C., Vauttier, S.: Search-based many-to-one component substitution. J. Softw. Maint: Res. Pract. **20**(5) (Sept. / Oct. 2008) 321–344
15. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline. Prentice-Hall. (1996)
16. Cheng, S.W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P.: Using architectural style as a basis for system self-repair. In: Proc. of 3rd WICSA Conf., Montreal, Canada (Aug. 2002) 45–59

Appendix E

A formal approach for managing component-based architecture evolution

This journal article [85] is to appear in Science of Computer Programming in October 2016. It is an extended, selected version of a paper presented at the 11th international symposium on Formal Aspects of Component Software (FACS 2014) [83]. It is the natural extension of the conference paper of Appendix D. It presents part of the work realized during the PhD of Abderrahman Mokni.

Software architectures are subject to several types of change during the software lifecycle (*e.g.*, adding requirements, correcting bugs, enhancing performance). The variety of these changes makes architecture evolution management complex because all architecture descriptions must remain consistent after change. To do so, whatever part of the architectural description they affect, the effects of change have to be propagated to the other parts.

The goal of this work is to provide support for evolving component-based architectures at multiple abstraction levels. Architecture descriptions follow the Dedal architectural model, the three description levels of which correspond to the three main development steps - specification, implementation and deployment. This article formalizes an evolution management model that generates evolution plans according to a given architecture change request, thus preserving consistency of architecture descriptions and coherence between them. To do so, it uses the B formal language and its companion B solver.

The approach is implemented as an Eclipse-based tool and validated with three evolution scenarios of a Home Automation Software example.

A formal approach for managing component-based architecture evolution

Abderrahman Mokni [a], Christelle Urtado[a], Sylvain Vauttier[a], Marianne Huchard[b], Huaxi Yulin Zhang[c]

[a] *LGI2P, Ecole Nationale Supérieure des Mines Alès, Nîmes, France*
[b] *LIRMM, CNRS and Université de Montpellier, Montpellier, France*
[c] *Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France*

Abstract

Software architectures are subject to several types of change during the software lifecycle (*e.g.* adding requirements, correcting bugs, enhancing performance). The variety of these changes makes architecture evolution management complex because all architecture descriptions must remain consistent after change. To do so, whatever part of the architectural description they affect, the effects of change have to be propagated to the other parts. The goal of this paper is to provide support for evolving component-based architectures at multiple abstraction levels. Architecture descriptions follow an architectural model named Dedal, the three description levels of which correspond to the three main development steps — specification, implementation and deployment. This paper formalizes an evolution management model that generates evolution plans according to a given architecture change request, thus preserving consistency of architecture descriptions and coherence between them. The approach is implemented as an Eclipse-based tool and validated with three evolution scenarios of a Home Automation Software example.

Keywords: architecture evolution, architecture analysis, evolution rules, formal models, abstraction level, evolution plans, MDE.

Email addresses: Abderrahman.Mokni@mines-ales.fr (Abderrahman Mokni), Christelle.Urtado@mines-ales.fr (Christelle Urtado), Sylvain.Vauttier@mines-ales.fr (Sylvain Vauttier), Marianne.Huchard@lirmm.fr (Marianne Huchard), yulin.zhang@u-picardie.fr (Huaxi Yulin Zhang)

1. Introduction

Component-based software development (CBSD) promotes a reuse-based approach to defining, implementing and composing loosely coupled independent software components into whole software systems [1]. While component reuse is crucial to shorten large-scale software systems development time, handling evolution in such processes is a significant issue [2]. Indeed, software systems have to evolve to extend their functionalities, correct bugs, improve performance and quality, or adapt to their environment. While unavoidable, software changes may engender several inconsistencies and system dysfunction if not analyzed and handled carefully. In turn, an ill-mastered evolution engenders software degradation, the loss of its evolvability and then its phase-out [3].

A famous problem of software evolution is software architecture erosion [4, 5]. It arises when modifications of the software implementation violate the design principles captured by its architecture. To increase confidence in reuse-centered, component-based software systems, all architecture descriptions must remain consistent and coherent with each other after every change.

While a lot of work has been dedicated to architectural modeling and evolution, there still is a lack of means and techniques to tackle architectural inconsistencies, and erosion in particular. Indeed, most existing approaches to architecture evolution hardly support the whole life-cycle of component-based software and only enable evolution of early stage models by propagating change impact to runtime models while evolution of runtime models are not fully dealt with, thus increasing the risks of architecture erosion.

This paper proposes an approach and its implementation to automatically manage component-based architecture evolution at multiple abstraction levels in a manner that preserves architecture consistency and coherence all along the software lifecycle. The approach is based on the Dedal [6, 7] architectural model that explicitly models architectures at three abstraction levels, each corresponding to one of the three major steps of CBSD – specification, implementation and deployment, thus granting a full evolution management process. Given a change request at any abstraction level, it transforms Dedal models into B formal models to analyze the requested change and generates an evolution plan that guarantees the consistency of architecture descriptions and the coherence between them. The proposed approach is centered on a formal evolution management model that includes the generated B mod-

els, the architecture properties to preserve and a set of evolution rules. It is implemented as an Eclipse-based tool that generates B models from diagrammatic Dedal models and uses our specific solver to resolve architecture evolution. The overall approach is illustrated with a Home Automation Software case-study.

The remainder of this paper outlines as follows: Section 2 presents the background of this work. Section 3 presents our proposal to tackle multi-level architecture evolution (*i.e.* the evolution of architecture definitions composed of multiple description levels) while Section 4 presents the implemented tool and experiments on three evolution scenarios. Section 5 discusses related work and finally, Section 6 concludes the paper and discusses future work.

2. Background

Our approach combines the use of Dedal to model software architectures and B to support automated analysis and verification. This section briefly introduces these languages.

2.1. The Dedal architecture model

2.1.1. Component-based software development by reuse

CBSD follows the *reuse-in-the-large* principle. Reusing existing (off-the-shelf) software components [8] therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [1]. Figure 1 illustrates our vision of such a development process which is classically divided in two:

- the component development process (referred to as software component development for reuse), which will not be detailed in the sequel. This development process produces components that are stored in repositories for later use by the software development process.
- the software development process (referred to as software development by component reuse) that describes how previously developed software components can be used for software development (and how this reuse impacts the way software is built).

Dedal is a novel architectural model and ADL [6, 7] that targets reuse-centered development. It covers the whole software development by component reuse life-cycle. The main idea of Dedal is to build a concrete architecture composed of stored and indexed components that are found in a

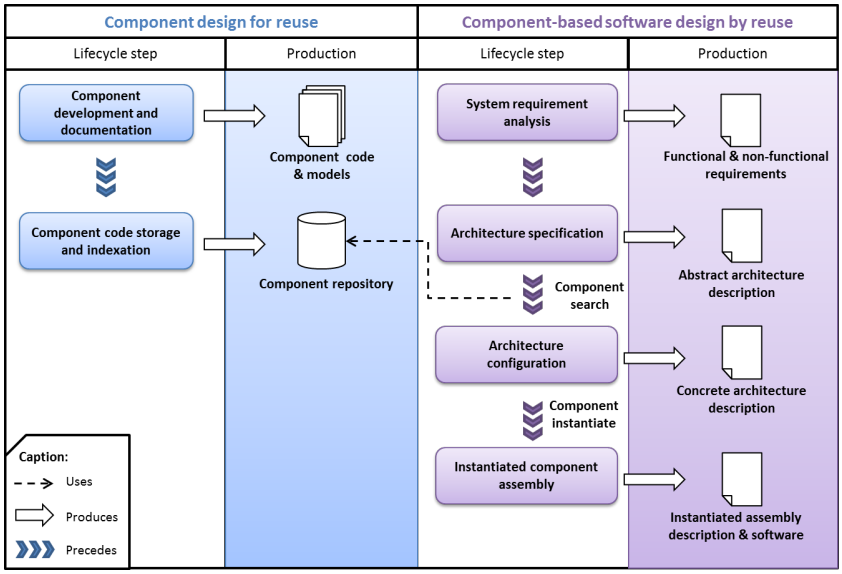


Figure 1: Dedal reuse-centered development process [7]

component repository as candidates to satisfy the design decisions specified in an intended architecture specification. The resulting concrete architecture can then be instantiated and deployed in multiple contexts. Therefore, Dedal proposes a three-step approach for specifying, implementing and deploying software architectures.

2.1.2. Dedal abstraction levels

To illustrate the concepts of Dedal, we propose to model a home automation software (HAS) that manages comfort scenarios, which automatically controls buildings' lighting and heating depending on time and ambient temperature. For this purpose, we propose an architecture with an orchestrator component that interacts with the appropriate devices to implement the desired scenario.

The *abstract architecture specification* is the first level of software architecture descriptions. It is abstract: it represents the architecture as imagined by the architect to meet the requirements of the future software. In Dedal, the architecture specification is composed of component roles, their connections and the expected global behavior. Component roles are abstract and partial component type specifications. Consequently, the provided interfaces of each role are to be connected to compatible required interfaces. Compo-

nent roles are identified by the architect in order to search for and select corresponding concrete components in the next step. Figure 2-a shows a possible HAS architecture specification. In this specification, five component roles are identified. A component playing the *HomeOrchestrator* role controls four components playing the *Light*, *Time*, *Thermometer* and *CoolerHeater* roles.

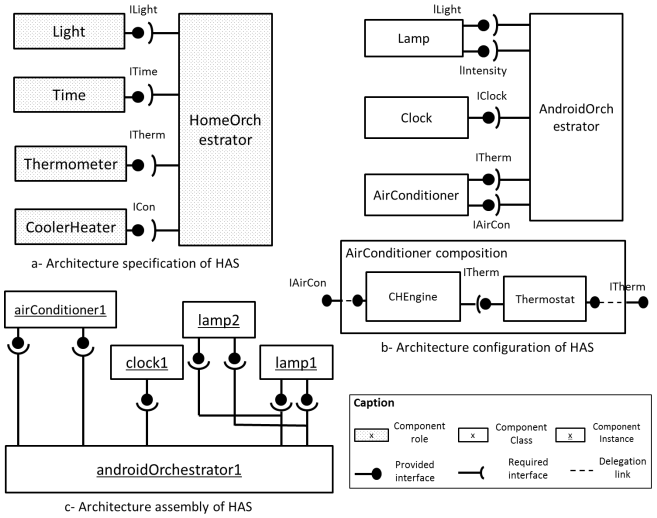


Figure 2: Architecture specification, configuration and assembly of the HAS

The *concrete architecture configuration* is an implementation view of software architectures. It results from the selection of existing component classes in component repositories. Thus, an architecture configuration lists and connects the concrete component classes that compose a specific version of the software. In Dedal, component classes can either be primitive or composite. A *primitive component class* encapsulates executable code. A *composite component class* encapsulates an inner architecture configuration (*i.e.* a set of connected component classes which may, in turn, be primitive or composite). A composite component class exposes a set of interfaces corresponding to the unconnected interfaces of its inner components. Figure 2-b shows a possible architecture configuration for the HAS example as well as an example of an *AirConditioner* composite component and its inner configuration. As illustrated in this example, a single component class may realize several roles from the architecture specification as with the *AirConditioner* component

class, which realizes both the *Thermometer* and *CoolerHeater* roles. Conversely, a component class may provide more services than those listed in (its role in) the architecture specification as with the *Lamp* component class which provides an extra service to control the intensity of light. These extra interfaces may be left unconnected.

The *instantiated architecture assembly* describes software at runtime and holds information about its internal state. The architecture assembly models an instantiation of its architecture configuration. It lists the instances of the component and connector classes that compose the deployed architecture at runtime and their assembly constraints (such as the maximum number of connected instances). *Component instances* document how the component classes from an architecture configuration are instantiated in the deployed software. Each component instance has an initial and a current state defined by a list of valued attributes. Figure 2-c shows an instantiated architecture assembly for the HAS example.

2.2. The B modeling language

B [9, 10] is a formal modeling language and a proof-based development method for software systems. The principle of such method is to start from a very abstract model of the system and then gradually refine it. Initially designed by Abrial in 1985 to specify critical systems, B was rapidly adopted by industry and used in many case studies such as the METEOR project [11] for controlling train traffic and the PCI protocol [12]. B is also widely used and studied in academia, mainly as a formal modeling language for verification, validation and model-checking.

2.2.1. Expressiveness and semantics

B is based on Zermelo-Fraenkel (ZF) set theory and first order logic language. The B notation is very similar to mathematical language and includes all standard logical connectors (*e.g.* $\wedge, \vee, \Rightarrow$), set-theoretic operations (*e.g.* $\in, \cup$), closure and specific relations like injective ($\rightarrowtail$), surjective ($\twoheadrightarrow$) and bijective ($\rightarrowtail\!\!\!\!\twoheadrightarrow$) functions. B also supports sequences and the basic boolean ($BOOL$), integer ($INTEGER$) and natural (NAT) types.

B specifications are composed of abstract machines similar to modules (*cf.* Figure 3). They are defined independently and can be reused as modules and refined to obtain more concrete models. An abstract machine is divided into a declarative part and a dynamic part. The declarative part

contains the declaration of sets (*SETS*), constants (*CONSTANTS*), variables (*VARIABLES*) which represent the state of the machine and invariant properties (*INVARIANT*) related to variables. Optionally, it is also possible to set definitions (*DEFINITIONS*) (like macros). Definitions are useful to define extensive sets and parametrized predicates and can be reused by invariants and operations. The dynamic part contains the initialization (*INITIALISATION*) of the machine as well as operations (*OPERATIONS*) over the state (variables) of the machine. The behavior of operations is explicitly defined in B using various constructs such as preconditions (*PRE P THEN S END*), bounded choice (*CHOICE S1 OR S2*) or non-determinism (*ANY v WHERE P THEN S END*). Post-conditions are expressed by substitutions that state the new assignments of the involved variables. Output variables may also be defined as values returned by operations.

MACHINE
name and eventually parameters
INCLUDES (optional)
imported specifications
(Static/declarative part)
SETS
declaration of abstract / enumerated sets
CONSTANTS
declaration of constants
PROPERTIES
constraints on constants
VARIABLES
declaration of variables (the machine state)
INVARIANT
declaration of invariant properties of the machine
DEFINITIONS (optional)
construction of formulas / sets using the variables of the machine
(Dynamic part)
INITIALISATION
initialization of the state of the machine (all declared variables)
OPERATIONS
definition of operations that modify the state of the machine

Figure 3: Structure of an abstract B machine

2.2.2. Tool support for B

B is supported by powerful tools like AtelierB [13], BToolkit [14] and the more recent Bware platform [15]. These tools focus on theorem-proving but they do not enable model-checking. ProB [16] was designed for this purpose. It is a model checker and animator for B models. It automatically generates counterexamples for given assertions by exhaustively exploring the model

(using state space exploration techniques). It also simulates the execution of operations on a given subset of the model and generates traces leading to some desired state. An API is also provided for developers to integrate the features of ProB in their tools.

2.3. Motivation and contribution

Component reuse helps decrease large-scale software systems time-to-market. Handling the evolution in such component-based software prevents architecture erosion and has long been identified and still remains an important thus difficult task [17, 18]. To tackle this issue, this paper proposes an approach to manage the evolution of component-based software architectures based on the three-level Dedal architecture model.

Dedal is tailored for reuse [6, 7] and provides as an original feature its three architecture definition levels. Indeed, specifications are the cornerstone of the concrete component search that is performed on component repositories to design, by reuse, the implementation of architectures. Along with Dedal configurations and assemblies, Dedal architecture definitions keep track of all the design decisions taken during the development process. This information is very useful to control evolution and evaluate its impact on the intentions of the architects. This is why Dedal is a choice ADL for architecture-based software evolution management.

The evolution process proposed here is driven by an evolution management model that captures changes initiated at any abstraction level, controls their impact to preserve / restore consistency and propagates them to other levels to maintain global coherence. This model is based on the B formal language which provides a rich and rigorous notation to formalize the architectural concepts and express properties over them. It supports automated analysis and model-checking thanks to the ProB tool.

In previous work [19, 20], we specified Dedal models using the B modeling language and proposed an evolution management model to enable the simulation, analysis and validation of evolution scenarios at any abstraction level using ProB. At that time, evolution was not yet automated since models were specified and evolved manually and separately. In the remainder, our approach integrating both Dedal and B to automatically manage component-based architecture evolution is presented. The automated Dedal to B transformation as well as a problem-specific B solver built on top of the ProB tool are the cornerstones of the contribution of this paper.

Using our problem-specific solver enables the automatic generation of evolution plans (sequences of change operations) to leverage the impact of a change request in a problem-specific manner and maintain the architecture descriptions coherent after change. The feasibility of our approach is demonstrated by experimenting on three evolution scenarios that each addresses change in a different abstraction level.

3. The formal evolution approach

This section presents our approach to formally handle the evolution of multi-level component-based architecture descriptions produced during software development. Its key idea is to use a B solver to automatically generate evolution plans that correspond to intended changes (*cf.* Figure 4).

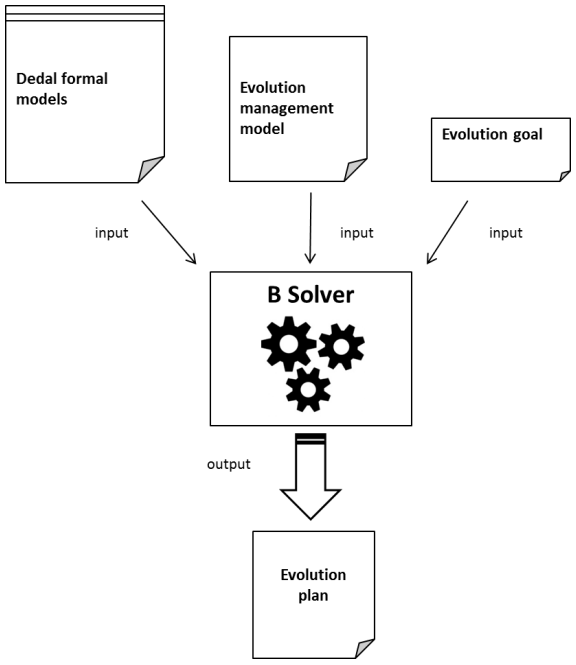


Figure 4: Evolution management approach

Given a model in an initial state, a set of state transition rules and a goal state, a B solver finds sequences of rules that reach the goal state or proves that the goal state cannot be reached (when it does not run out of time or resources because of high computational complexity).

A first requirement is thus to transform the Dedal models produced during development into B models that can be used as an input for the solver. The principles of this transformation are detailed in Section 3.1. Architecture evolution operations along with validation properties must also be expressed as a set of rules. The resulting *Evolution Management Model* is presented in Section 3.2. Finally, initiated architecture changes must be described as goal states, as explained in Section 3.3. With these inputs, a B solver can then find an evolution plan (a sequence of rules) that achieves the intended change (reaches the goal state) while preserving the coherence of the architecture definition (enforcing properties), as presented in Section 3.4.

3.1. Dedal to Formal Dedal transformation

Dedal models need to be translated into B models, so that a B solver can calculate modifications and evaluate properties on the resulting formal architecture descriptions. Defining this transformation amounts to formalize in B the concepts of the Dedal meta-model (*cf.* Figure 6). This way, any instance of the Dedal meta-model can be transformed into an equivalent instance of the Formal Dedal meta-model (*cf.* Figure 5).

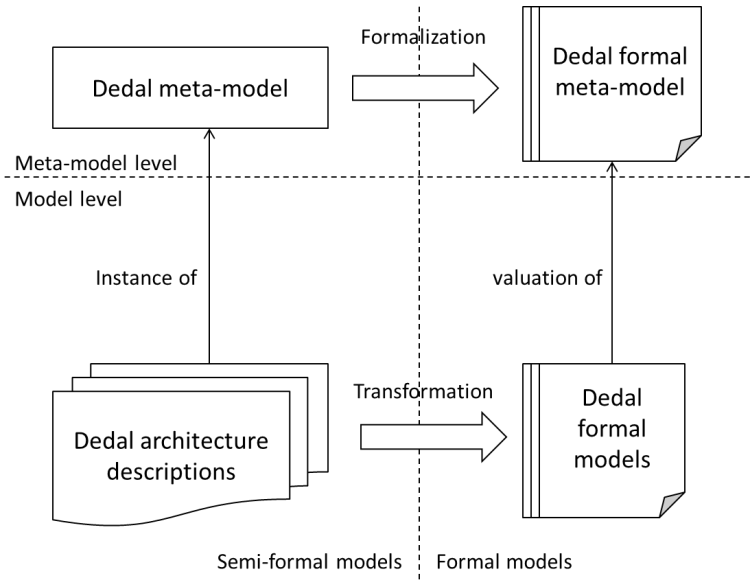


Figure 5: Dedal to Formal Dedal transformation

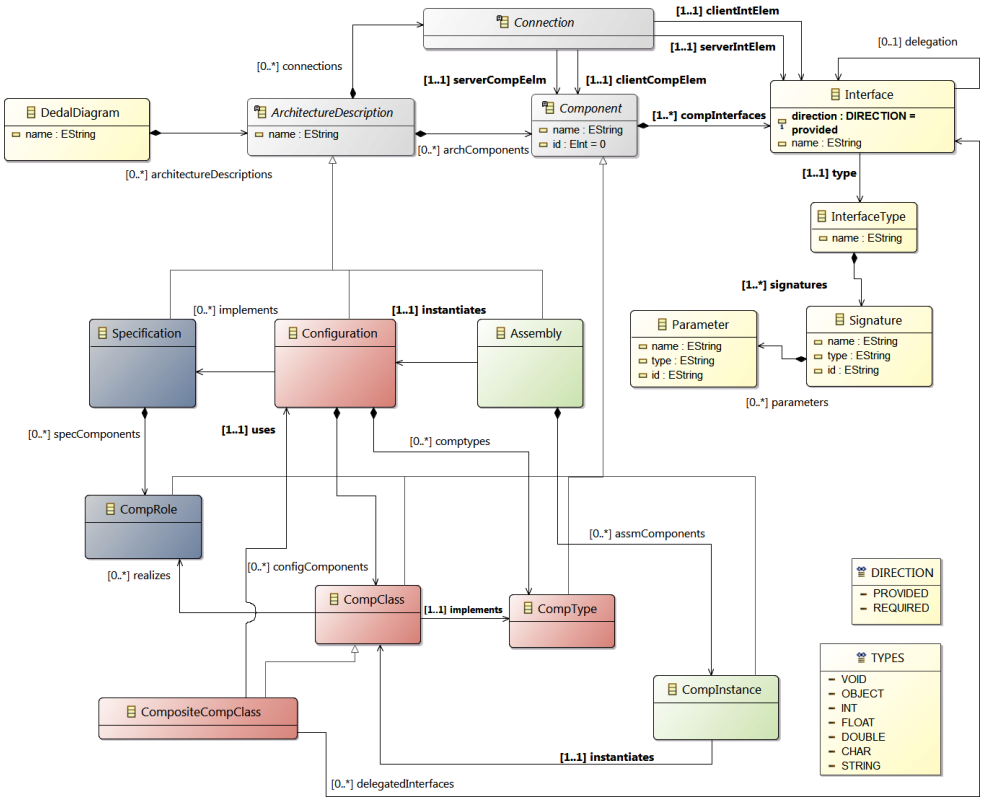


Figure 6: Dedal meta-model

A meta-class is usually mapped to a B variable typed by an abstract B set while an association relation is translated into a B relation. For instance, Figure 7 presents the formalizaton of the *Component* and *Interface* meta-classes and their *compInterfaces* association.

SETS
$COMPS; INTERFACES$
VARIABLES
$component, interface, comp\_interfaces$
INVARIANT
$component \subseteq COMPS \wedge$
$interface \subseteq INTERFACES \wedge$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}_1(interface)$

Figure 7: Formalization of meta-classes and associations in B

The *Component* and *Interface* meta-classes are respectively mapped to the *component* and *interface* variables and typed with the *COMPS* and *INTERFACES* abstract sets. Their *compInterfaces* association holding a one-to-many relation is translated into an injective function between the *component* variable and a non-empty set of interfaces: $\mathcal{P}_1(interface)$.

The whole Dedal meta-model formalization results in four main B machines (extracts of which are shown in Figure 8).

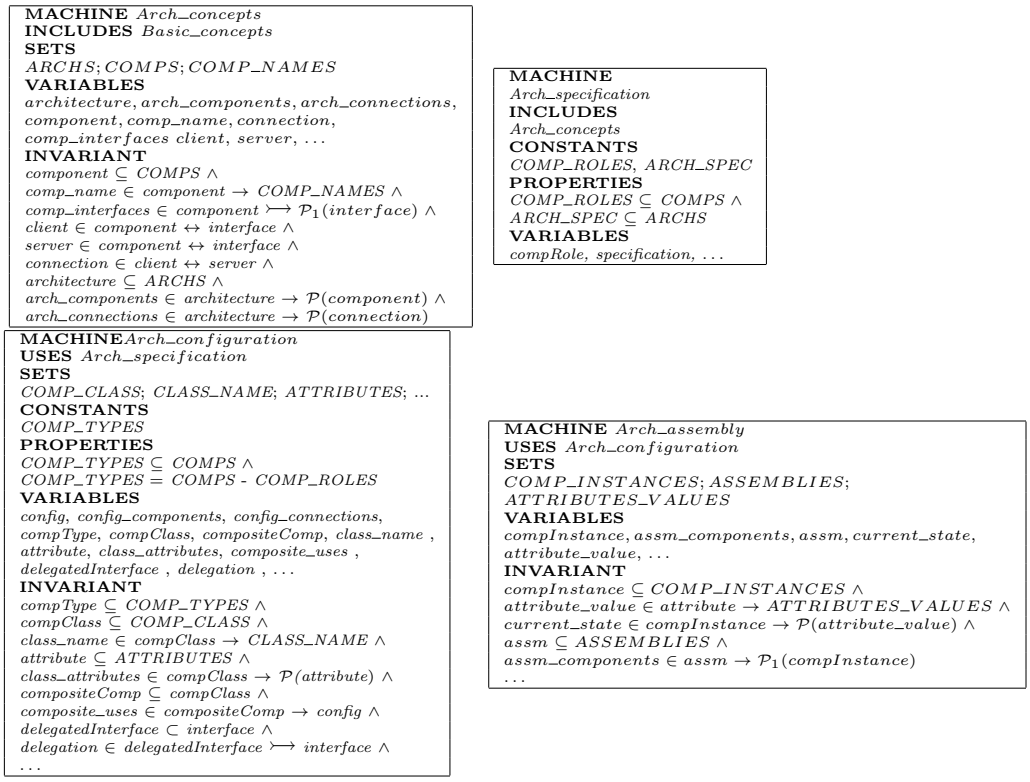


Figure 8: Overview of the Dedal formal meta-model

A generic *Arch_concepts* machine helps define the three specific *Arch_specification*, *Arch_configuration* and *Arch_assembly* machines that each correspond to one of the three architecture description levels of Dedal. *Arch_concepts* covers the generic concepts of a software architecture (corresponding to the abstract *Component*, *Connection*, and *ArchitectureDescription* meta-classes). It includes an inner *Basic_concepts* machine that contains definitions for the finer-grained architectural elements like *Interface*, *InterfaceType*, *Signature*

or *Parameter* meta-classes.

These generic definitions are reused in the three specific machines. For instance, in the *Arch_specification* machine, component roles are defined as a subset of components: $COMP\_ROLES \subseteq COMPS \wedge compRole \subseteq COMP\_ROLES$.

This corresponds to the inheritance relation between the *Component* and *CompRole* meta-classes. Consequently, all relations defined for the *component* set (such as *comp_interfaces*) also stand for the *compRole* set.

The abstract B machines define a formal meta-model that can be instantiated (concrete values are given to their variables) in order to generate a Formal Dedal model. The latter is then used as an input for the B solver.

3.2. The evolution management model

The evolution management model is composed of generic evolution rules that are used by the solver to find evolution plans satisfying given evolution goals. It consists in a B machine that defines the rules and properties that respectively enable the simulation and validation of architecture evolution at the three abstraction levels (*cf.* Figure 9).

MACHINE
EvolutionManager
INCLUDES
Arch_specification, *Arch_configuration*, *Arch_assembly*
SETS
/*Enumerated set to indicate the level of change*/
$CHANGE\_LEVEL = \{eLevel, specLevel, configLevel, asmLevel\}$
VARIABLES
/*Variable to control the level of change*/
changeLevel, ...
DEFINITIONS
/*Consistency and coherence properties*/
...
$global\_consistency == spec\_consistency \wedge config\_consistency \wedge assm\_consistency$
$global\_coherence == specConfigCoherence \wedge configAssmCoherence$
/*GOAL is the predicate given to the solver to find an evolution plan satisfying it*/
$GOAL == global\_consistency \wedge global\_coherence \wedge \ldots$
INITIALISATION
/*Initialization is used to set the initial level of change and the initiated change*/
...
OPERATIONS
/*Initialization operations*/
...
/*Evolution rules(control the architecture manipulation operations) ***/**
...
END

Figure 9: The *EvolutionManager* machine

Its main elements are detailed in the following subsections.

3.2.1. Evolution rules

Evolution rules are operations that control the access and the impact of architecture manipulation operations in order to manage evolution and generate consistent evolution plans (*cf.* Figure 10). Each evolution rule embeds a corresponding architecture manipulation operation that handles the actual modification of the model, not taking into account the context of the current evolution plan.

output ← evolutionRuleName(targetArchitecture, artifacts) =
PRE
$initialization = true \wedge$
$changeLevel = currentChangeLevel \wedge$
$artifacts \notin addedArtifacts \cup deletedArtifacts \wedge$
$manipulationOperationPrecondition$
THEN
/* execute manipulationOperationName(targetArchitecture, artifacts),
update the sets of added artifacts and deleted artifacts,
set the value of output parameters */
END

Figure 10: Schema of an evolution rule

The evolution rule preconditions act as a primary filter for model manipulation operations. Initialization preconditions check that all the model initialization operations have completed before starting calculating evolution plans. Initialization includes calculating and checking relations between architecture elements, such as compatibility and substitution between components and interfaces. Change level preconditions restrict access to the operations related to the current level of change (evolution is managed on one level at a time). History preconditions prevent operations that may generate cycles and then decrease the efficiency of the solver. For instance, deleting and adding the same artifact several times is unnecessary during an evolution process. Similarly, removing an added artifact results in a null operation that may be avoided. History consists of two sets: one for added artifacts and the other for deleted ones. Evolution rules also inform the solver about the artifacts that have to be manipulated after the last executed change operation. This information is used as a heuristic to increase the efficiency of the solver. Heuristics are further discussed in Section 4.1.2.

Figure 11 gives the definition of the evolution rule that controls the role addition operation. This rule is enabled when evolution is handled at the

specification level, after initialization, provided that the role has not yet been added or previously removed. If so, the precondition of the role addition operation is checked and, when it is verified, the operation is executed. Finally, the set of added component roles (*addedRoles*) is updated and the output is set to the added component role (*newRole*).

output ← **mng_addRole**(*spec*, *newRole*) =
PRE
/*Initialization precondition*/
    initialisation = **TRUE** ∧
/*Change level precondition*/
    changeLevel = *specLevel* ∧
/*Precondition to avoid cycles (inverse operation)*/
    newRole ∉ (*deletedRoles* ∪ *addedRoles*) ∧
/*Precondition of the role addition operation*/
    roleAdditionPrecondition
THEN
/*Access to role addition operation*/
    addRole(*spec*, *newRole*) ||
    addedRoles := *addedRoles* ∪ {*newRole*} ||
    outpout := *newRole*
END;

Figure 11: The component role addition evolution rule

3.2.2. Model manipulation operations

A model manipulation operation is an operation that changes a target software architecture by the deletion, addition or substitution of one of its elements (components and connections). They are composed of three parts:

- the operation signature that defines the operation name and its arguments,
- preconditions that are related to the architectural model (*e.g.* a precondition that checks if substitutability between two component classes holds),
- actions (called substitutions in B) that update a set of variables related to the architectural model (*e.g.* the set of components of the architecture).

Architecture specification evolution. Evolving an architecture specification is usually a response to a new software requirement. For instance, the architect may need to add new functionalities to the system and hence add some new roles to the specification. Moreover, a specification may also be modified during the change propagation process to preserve coherence and keep an up-to-date specification description of the system that may be implemented in several ways. The proposed manipulation operations related to the specification level are the addition, deletion and substitution of a component role and the addition and deletion of connections. Figure 12 presents the definition of the role addition operation as an example of an architecture specification manipulation operation. Its precondition first checks that arguments are soundly typed and then that the chosen role does not already belong to the architecture specification and will not name clash. Its actions update the set of component roles of the architecture specification, along with the sets of connected provided and required interfaces (respectively *spec_components*, *spec_servers* and *spec_clients*). Indeed, as only effectively used elements are defined at specification level, every interface must be connected.

addRole(*spec*, *newRole*) =
PRE
$spec \in arch\_spec \land newRole \in compRole \land newRole \notin spec\_components(spec) \land$
/* *spec* does not contain a role with the same name*/
$\forall\ cr.(cr \in compRole \land cr \in spec\_components(spec)$
$\Rightarrow comp\_name(cr) \neq comp\_name(newRole))$
THEN
    spec_servers(*spec*) := *spec_servers*(*spec*) ∪ servers(newRole) ||
    spec_clients(*spec*) := *spec_clients*(*spec*) ∪ clients(newRole) ||
    spec_components(*spec*) := *spec_components*(*spec*) ∪ {*newRole*}
END;

Figure 12: The component role addition manipulation operation

Architecture configuration evolution. Change can be initiated at the configuration level, for example when new versions of software component classes are released or when component classes are not available anymore. Otherwise, an implementation may also be impacted by change propagation either from the specification level, in response to new requirements, or from the assembly level, in response to a dynamic change of the system. Indeed, a configuration may be instantiated several times and deployed in multiple contexts. Figure 13 presents the component class substitution operation as an example of an architecture configuration manipulation operation.

replaceClass(*config*, *oldClass*, *newClass*) =
PRE
oldClass ∈ *compClass* ∧ *newClass* ∈ *compClass* ∧ *config* ∈ *configuration* ∧
oldClass ∈ *config_components*(*config*) ∧
/* The old component class can be substituted for the new one
(verified by the component substitution rule)*/
newClass ∉ *config_components*(*config*) ∧ (*oldClass*, *newClass*) ∈ *class_substitution*
THEN
config_components(*config*) := (*config_components*(*config*) - {*oldClass*}) ∪ {*newClass*}
END

Figure 13: The component class substitution manipulation operation

Besides checking the type of the arguments, its precondition verifies that the new component class does not already belong to the configuration and can be a substitute for the old component class (using the relations calculated during initialization). When the precondition is verified, the set of component classes composing the configuration is updated. As compared to the role addition operation presented in previous section, there is no need to update the sets of client and server interfaces (connected required and provided interfaces) here, as substitution must preserve the connections of the replaced component class (*see* § 3.2.3 for deeper insight about substitution rules).

Architecture assembly evolution. Since the architecture assembly represents the software at runtime, managing the assembly level relates to dynamic evolution issues. Indeed, some software systems have to be self-adaptive to keep providing their functions despite environmental changes (*e.g.* lack of resources, failures, user requests). Dealing with unanticipated changes is one of the most important issues in software evolution. This issue is handled by the evolution manager which monitors the execution state of the software through its corresponding formal model. It then triggers the assembly evolution rules to restore consistency and coherence when needed. The assembly manipulation operations include component instance addition, component instance removal, component instance substitution and component instance connection / disconnection. Figure 14 gives the definition of the component instance addition as an example of an assembly manipulation operation. After checking the types of the arguments, the precondition verifies that the instance corresponds to the chosen component class, that it does not already belong to the assembly and that another instance of the class can be added in the assembly. It also verifies that the chosen initial state is valid.

deployInstance(*asm*, *inst*, *class*, *state*) =
PRE
asm ∈ *assembly* ∧ *class* ∈ *compClass* ∧
/* The instance is a valid instantiation of the chosen component class*/
inst ∈ *compInstance* ∧ *class* = *comp_instantiates*(*inst*) ∧ *inst* ∉ *assm_components*(*asm*) ∧
/* The state given to the instance is a valid value assignment of its attributes of the instantiated component class*/
state ∈ $\mathcal{P}$ (*attribute_value*) ∧ **card**(*state*) = **card**(*class_attributes*(*class*)) ∧
/* The maximum number of allowed instances of the given component class is not already reached*/
nb_instances(*class*) < *max_instances*(*class*)
THEN
/*initial and current state initialization*/
initial_state(*inst*) := *state* ||
current_state(*inst*) := *state* ||
/*updating the number of instances and the assembly architecture*/
nb_instances(*class*) := *nb_instances*(*class*) + 1 ||
assm_components(*asm*) := *assm_components*(*asm*) ∪ {*inst*} ||
assm_clients(*asm*) := *assm_clients*(*asm*) ∪ *clients*(*inst*)
END;

Figure 14: The component instance deployment manipulation operation

When executed, the operation adds the instance in the assembly, updates the count of instances of the component class and updates the set of client interfaces. The set of server interfaces will be updated later, as client interfaces are automatically connected by the evolution manager to maintain the consistency of the assembly (*see* § 3.2.3).

Manipulation operations constitute the dynamic aspect of the architectural formal models. They enable to change the state of a model which must therefore be validated thanks to consistency and coherence properties exposed in the following sections.

3.2.3. Consistency properties

Consistency properties maintain the correctness of each architecture description level during the evolution process. Taylor *et al.* [21] define consistency as an internal property intended to ensure that different elements of an architecture model do not contradict one another. They point out five kinds of inconsistencies that may occur in architecture models: name, interface, behavior, interaction and refinement. Our consistency properties deal with the following inconsistencies:

- *Name consistency* ensures that each component holds a unique name to avoid conflicts when selecting components.

- *Interface consistency* ensures that all architecture connections are correct (*i.e.* a required interface is always connected to a compatible provided interface).

- *Interaction consistency* ensures that the architecture realizes its functional objectives (components are able to soundly cooperate through their connected interfaces). In our approach, this property is implemented as a verification that each required interface is connected to a compatible provided one. Moreover, in architecture specifications, all server interfaces must also be connected (no unused feature is described at this level). Besides, every architecture definition must be composed of a connected graph, so that no part of the architecture is isolated.

Behavior consistency is out of the scope of the work presented in this paper which only considers static type definitions, for now. *Refinement consistency* is handled separately by our coherence properties (*cf.* Section 3.2.4).

As an example, the formalization of our interface consistency property is presented in Figure 15.

$$
\begin{array}{l}
\forall\ (cl,\ se).(cl \in client \wedge se \in server \Rightarrow \\
\quad ((cl,\ se) \in connection \Rightarrow \\
\quad \exists\ (C_1\ ,\ C_2\ ,\ int_1\ ,\ int_2\ ).(C_1 \in component \wedge C_2 \in component \wedge C_1 \neq C_2 \wedge \\
\quad int_1 \in interface \wedge int_2 \in interface \wedge cl = (C_1\ ,\ int_1) \wedge se = (C_2\ ,\ int_2) \wedge \\
\quad (int_1\ ,\ int_2) \in int\_compatible)))
\end{array}
$$

Figure 15: Interface consistency property

This property states that a required (client) interface is properly connected to a provided (server) interface when these two interfaces belong to different components and have compatible types.

Consistency properties are based on commonly adopted syntactic typing rules that state compatibility and substitution between finer grained entities such as components and interfaces. These rules transpose the well studied typing principles used in the object-oriented paradigm to the component-oriented paradigm. As usual, the main principle is that a component that belongs to a subtype can substitute for a component that belongs to a supertype (*i.e.* be connected at the same place in the same architecture). This entails that a component subtype must define a set of interfaces that can replace all the interfaces defined in its supertype (identical interfaces

or interfaces belonging to subtypes). Moreover, a component subtype cannot define extra required interfaces, as they correspond to extra connection requirements that break the substitution guarantee with the supertype. Conversely, extra provided interfaces can be defined in a subtype as they do not imply mandatory extra connections.

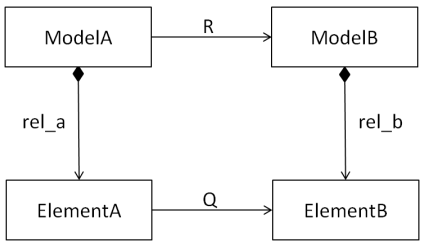
Comparing component types thus amounts to comparing interface types. Interface type hierarchies are built with respect to the same substitution principle: an interface subtype must define a set of operations that can replace those of its supertypes. Usual specialization rules are applied to provided interface types, that are comparable to object types. A provided interface subtype must define at least the same operations as its supertypes or specialized operations that can replace them. Classically, an operation specializes another one when it has the same name, a contravariant set of input parameters (at most as many parameters, with identical or more generic data types) and a covariant set of output parameters (at least as many parameters, with identical or more specific data types). With these rules, it is always possible to call a more specialized operation with the input values of a more generic one and then to use the output values of the more specialized operation in place of the output value of the more generic one.

Regarding required interfaces, opposite specialization rules are used. Indeed, a required interface corresponds to dependencies. Thus, a required interface subtype cannot define more operations than its supertypes, in order not to add extra dependencies. It cannot define less operations either, as this can impair interactions with other components. A required interface subtype must then implement the same operations as its supertypes, or more generic operations (*i.e.* operations with the same name, at least as many input parameters of identical or more specific data types and at most as many output parameters with identical or more generic data types). Requiring more generic operations than its supertypes, a more specialized required interface can replace a more generic required interface. Dedal typing rules are discussed and detailed in previous work [22].

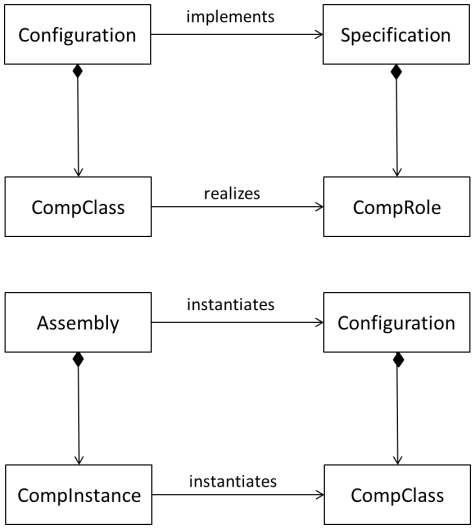
Compatibility is calculated thanks to the aforementioned typing rules. Basically, a required interface is compatible with a provided interface when they have the same type (*i.e.* are defined by the same set of operations). The required interface is also compatible with a provided interface that belongs to a subtype of its type (because of the substitution principle). Compatibility rules are also detailed in [22].

3.2.4. Coherence properties

Coherence properties prevent architecture erosion (mismatches between the different description levels) so as to maintain the global correctness of architecture definitions. Coherence properties maintain the relations that must exist between the specification, configuration and assembly defining an architecture (*cf.* Figure 16-b): its configuration must be a valid implementation of its specification; its assembly must be a valid instance of its configuration.These relations between description levels rely on typing relations between their composing elements. The component classes composing the configuration of an architecture must implement the component roles of its specification. In the same way, the component instances composing its assembly must be valid instances of the component classes of its configuration. This relates to a generic principle (*cf.* Figure 16-a) that a relation between two kinds of models implies a relation between their composing elements (and possibly reciprocally under restrictive conditions). For instance, a model can be considered as a specialization of another model only when its composing elements specialize the elements of the other model.



(a) inter-model coherence relation generic principle



(b) coherence relations between the Dedal architecture levels

Figure 16: Coherence relations between architecture levels

The generic principle can be formalized by the generic coherence rule

depicted in Figure 17.

$$
\begin{array}{l}
coherence(model_A, elem_A, model_B, elem_B, rel_a, rel_b, R, Q) == \\
\forall (M_a, M_b).(M_a \in model_A \wedge M_b \in model_B \Rightarrow ((M_a, M_b) \in R \\
\quad \Leftrightarrow \\
(\forall\, e_b.(e_b \in elem_B \wedge (M_b, e_b) \in rel_b \Rightarrow \\
\quad \exists\, e_a.(e_a \in elem_A \wedge (M_a, e_a) \in rel_a \wedge (e_a, e_b) \in Q)))))
\end{array}
$$

Figure 17: Generic coherence rule

In our work, two properties are defined in the *Evolution Management Machine* to assert the coherence of an architecture definition : coherence between configuration and specification and coherence between assembly and configuration.

Coherence between configuration and specification. A specification is a formal description of software requirements that is used to guide the search for suitable concrete component classes to implement the software. An architecture configuration is coherent with a specification when two properties hold:

- all component roles from the specification are realized by component classes in the configuration. This results in a many-to-many relation as several component roles may be realized by a single component class while, conversely, several component classes may be needed to realize a single role. Using the generic coherence rule (*cf.* Figure 17), this first property can be expressed as shown in Figure 18.

$$
\begin{array}{l}
implements \in configuration \leftrightarrow specification \wedge \\
coherence(configuration, compClass, specification, compRole, \\
\qquad config\_components, spec\_components, implements, realizes)
\end{array}
$$

Figure 18: Implementation coherence property using the generic rule

  To illustrate the instantiation of the generic coherence rule, we give the expansion of the implementation coherence property in Figure 19. In the remainder (Figure 20 and Figure 21), only the generic coherence rule is used.

- each connected provided (*server*) interface in the configuration is defined in the specification. This prevents having a configuration that implements extra functions not specified at the higher level which leads to architectural drift or erosion (*cf.* Figure 20).

$implements \in configuration \leftrightarrow specification \wedge$
$\forall\ (Conf,\ Spec).(Conf \in configuration \wedge Spec \in specification \Rightarrow$
$(Conf,\ Spec) \in implements$
$\Leftrightarrow$
$\forall\ CR.(CR \in compRole \wedge CR \in spec\_components(Spec) \Rightarrow$
$\exists\ CL.(CL \in compClass \wedge CL \in config\_components(Conf) \wedge$
$(CL, CR) \in realizes)))$

Figure 19: Implementation coherence property (expanded)

$conform \in specification \leftrightarrow configuration \wedge$
$coherence(configuration, server, specification, server,$
$\quad config\_servers, spec\_servers, conform, int\_substituion')$
where :
$(s, s') \in int\_substitution' \Leftrightarrow (serverInterfaceElem(s), serverInterfaceElem(s')) \in int\_substitution$

Figure 20: Provided interface connection coherence property

Coherence between assembly and configuration. As the definition of an assembly is not obtained from a configuration by an instantiation process (assemblies are defined at design-time), coherence between assembly and configuration descriptions must be checked *a posteriori* explicitly. An assembly is coherent with a configuration when every class of the configuration is instantiated at least once in the assembly and, conversely, every component instance in the assembly is a valid instance of a component class of the configuration (*cf.* Figure 21).

$instantiates \in assembly \rightarrow configuration \wedge$
$coherence(assembly, compInstance, configuration, compClass,$
$\quad assm\_components, config\_components, instantiates, comp\_instantiates)$
$\wedge$
$coherence(configuration, compClass, assembly, compInstance,$
$\quad config\_components, assm\_components, instantiates^{-1}, comp\_instantiates^{-1})$
where:
$instantiates^{-1}$ and $comp\_instantiates^{-1}$ are the respective reverse relations of $instantiates$ and $comp\_instantiates$

Figure 21: Configuration instantiation coherence property

3.3. Evolution goal

The evolution goal (GOAL) consists in a predicate definition that the solver will attempt to satisfy by searching for a valid sequence of evolution

rules (evolution plan) to execute on the architecture. The evolution goal consists of a static and a variable part. The static part contains all the consistency (*global_consistency*) and coherence (*global_coherence*) properties: the calculated evolution plan must maintain the validity of the architecture. The variable part contains the arguments of the initiated change: the evolution plan must achieve the intended change. For example, if the initiated change consists in the addition of a component role *cr* in a specification *spec*, the evolution goal would be the following:

$$GOAL == global\_consistency \wedge global\_coherence \wedge cr \in spec\_components(spec)$$

3.4. Evolution plan generation

Our evolution process distinguishes two kinds of change: initiated change and triggered change. *Initiated changes* have an external source: they originate from a user action or from the execution environment. *Triggered changes* are induced by the evolution manager to restore architecture consistency at each level (they are called *local changes*) and / or global architecture coherence (they are called *propagated changes*), after they have been impacted by an initiated change.

Evolution is handled as a three step process (*cf.* Figure 22).

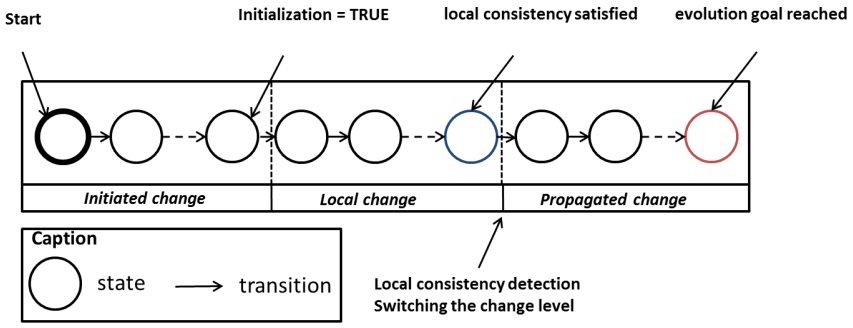


Figure 22: Evolution plan generation process

First, the initiated changes that compose a change request are all processed. These changes all affect a given level of architecture description (called the changed architecture level). In a second step, the impact of these initiated changes are calculated at the changed architecture definition level, thanks to the consistency properties. Maintaining consistency may imply additional (triggered) changes. Finally, the impact of these changes on the

other architecture definition levels are calculated thanks to the coherence properties. Maintaining coherence may also imply additional (propagated) changes on the other architecture definition levels.

4. Implementation and experimentation

To support our approach, we have implemented DedalStudio, a CASE tool which provides a Dedal modeler, a Formal Dedal generator and an evolution manager based on a solver. Three experiments are then presented in this section to assert the feasibility of our formal evolution approach. Each evolution scenario illustrates a change propagation issue that starts at a different abstraction level, in order to cover the three kinds of multi-level evolution: top-down, bottom-up and mixed. Finally, we evaluate the performance of our solver on the basis of the three experiments.

4.1. DedalStudio

4.1.1. Architecture of the tool suite

To validate our approach, we have implemented DedalStudio, an Eclipse-based modeling and evolution management environment for Dedal. The tool architecture is shown in Figure 23.

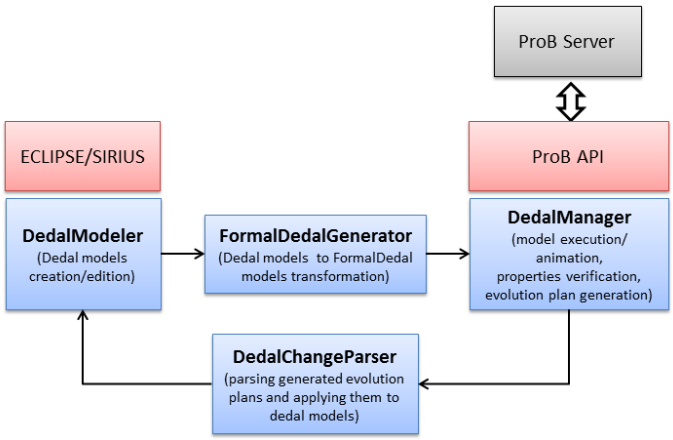


Figure 23: Architecture of the Dedal modeling and evolution management environment

DedalStudio enables the creation of architecture definitions, using a graphical concrete syntax designed for the Dedal meta-model, composed of Specification Diagrams (SD), Configuration Diagrams (CD) and Assembly Diagrams (AD). The diagram editor (*DedalModeler*), shown in Figure 24 is

based on SIRIUS [1], a generic platform that enables the creation of graphical modeling tools on top of EMF (Eclipse Modeling Framework) [2]. The *FormalDedalGenerator* creates Formal Dedal models corresponding to Dedal diagrams. The *DedalManager* handles the evolution process and the generation of evolution plans. It implements a customized solver built upon the ProB API [3] that enables the animation and model-checking of B models. Finally, the *DedalChangeParser* parses the generated evolution plans and apply the manipulation operations on the Dedal models. All theses tools, except for *DedalModeler* which is targeted to the architect, are fully automatic.

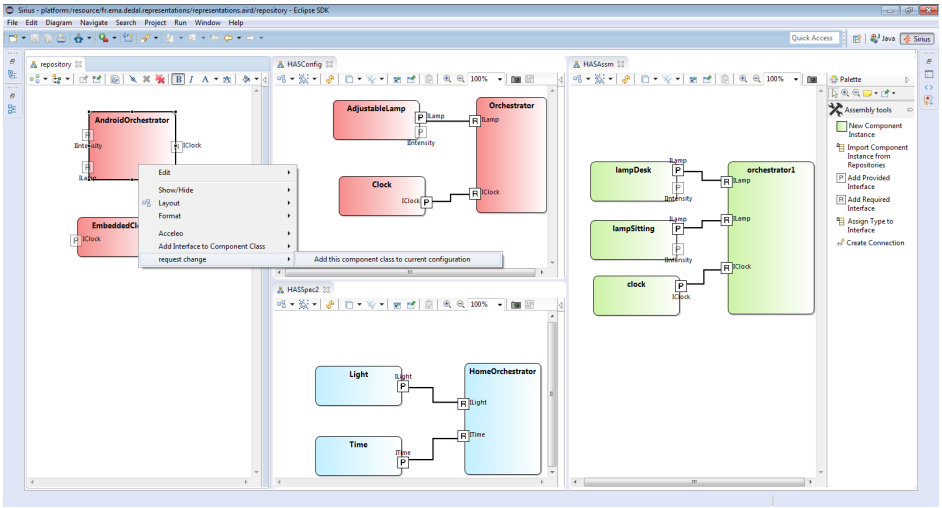


Figure 24: The DedalModeler tool

4.1.2. The DedalManager solver

Evolution management starts when a change to the architecture model is requested (for instance, a component class addition is requested in the configuration). The *DedalManager* receives the request, identifies the change level and deduces the evolution goal. It then invokes its solver, that conforms to the design principles presented in Section 3. The resolution algorithm implemented in the solver explores the search space to find a sequence of evolution

[1] https://eclipse.org/sirius/
[2] https://eclipse.org/modeling/emf/
[3] http://stups.hhu.de/ProB/w/ProB_Java_API

rules leading to the chosen goal. If a solution is found, the *DedalManager* generates an evolution plan that can then be committed by user. Otherwise (*i.e.* in case of failure), the *DedalManager* rejects the change request.

In previous work [20], we have made an evaluation of the performances of the ProB solver to generate evolution plans by state space exploration. The tested strategies were Depth-First (DF), Breadth-First (BF) and mixed (DF/BF) [23]. In most cases DF performed best, better than DF/BF and BF. The ProB solver, however, is general-purpose and increasing resolution time (over 3 minutes) is necessary when models become complex. To try and overcome this problem, this paper proposes an alternative: the implementation of a customized solver, using the API provided with ProB. It also consists in a depth-first search algorithm but enhanced with two specific heuristics: the artifact-oriented heuristic and the operation-oriented heuristic.

The artifact-oriented heuristic. The idea of artifact-oriented heuristic is to prioritize the operations manipulating the artifacts that are more likely to satisfy the evolution goal (thereafter called the main artifacts). For instance, adding a new component usually entails several connection operations on that component to restore architecture consistency. Main artifacts are determined at each iteration of the search process by the output of last executed evolution rule.

The operation-oriented heuristic. The operation-oriented heuristic adopts an opposite point of view. It delays the use of operations that engender unsatisfied dependencies between the components of the architecture and hence more evolution operations to be found in order to reestablish architecture consistency. Addition operations are the most concerned ones. They are therefore ordered as the least priority operations while performing the search process.

The search algorithm. Listing 1 describes the search algorithm of our customized solver. Lines 1–14 define and initialize the main variables of the algorithm. `Transitions` refers to the set of all the evolution rules instances in the current state of the architecture model. The set of already explored transitions is stored in `visited`, in order to avoid cycles in the search process. The current sequence of executed transitions is stored in `pl`, to collect the candidate evolution plan. The traversal of the search graph is handled by `stack`. At each step of the search process, the set of all the enabled transitions (*i.e.* the evolution rule instances whose preconditions are verified) is

pushed on the stack in order to explore them in the next steps. Transitions are pushed on the stack along with the current state of the architecture model and the current evolution plan. This enables to backtrack to previous nodes in the search graph and explore other paths when dead ends are reached. The main artifact `a` is used in the evaluation of the artifact-oriented heuristics. The `initialMainArtifact` references the artifact modified by the initiated change. It is calculated from the post-conditions of the corresponding operations.

```
1  // initialisation step
2  s = initialState;
3  a = initialMainArtifact;
4  pl = null;
5  stack = null;
6  visited = ∅;
7  enabledTransitions = {e_i ∈ Transitions where pre(e_i) == true};
8  priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
9  lowpriorTransitions = ∅;
10 enabledTransitions = enabledTransitions - priorTransitions;
11
12 // organizing stack
13 stack.push(s, pl, enabledTransitions);
14 stack.push(s, pl, priorTransitions);
15
16 // starting forward, DF search
17 while (stack ≠ ∅)
18  {
19  (s, pl, e_i) = stack.pop();
20  if ((s, e_i) ∉ visited)
21   {
22   visited = visited ∪ {(s, e_i)};
23   s = execute(e_i);
24   pl = pl+e_i;
25   if (goal == true) return pl;
26   a = output(e_i);
27   enabledTransitions = {e_i ∈ Transitions where pre(e_i) == true};
28   priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
29   lowpriorTransitions = {e_i ∈ enabledTransitions where h2(e_i) == true};
30   enabledTransitions = enabledTransitions - (priorTransitions ∪
         lowpriorTransitions);
31   stack.push(s, pl, lowpriorTransitions);
32   stack.push(s, pl, enabledTransitions);
33   stack.push(s, pl, priorTransitions);
34   }
35  }
36 return null; // no solution for this change request
```

Listing 1: Search algorithm of our specific solver

At each iteration of the search process (lines 17–33), the top of the `stack` is popped (line 19), setting a context consisting of an architecture model state (`s`), an evolution plan (`pl`) and an enabled transition (e_i). If the transition

has already been visited from this state (line 20), another context is popped from the `stack` (this happens when a state can be reached by several paths of the search tree). If the transition has not been explored, it is listed as `visited` (line 22) and executed (line 23), updating the state of the architecture model. The last executed transition is appended to the evolution plan (line 24). If the `goal` is satisfied, an evolution plan has been found and it is returned (line 25). Otherwise, the set of the enabled transitions in the current state is calculated (line 27) as is the set of higher priority enabled transitions (line 28) based on the artifact-oriented heuristic (`h1`). This uses the main artifact defined as the output of the last executed transition (line 26). The set of lower priority enabled transitions is also calculated (line 29), based on the operation-oriented heuristic (`h2`). This enables to push on the `stack` the enabled transitions to be explored depending on the priority determined by our heuristics (lines 31–33). The use of a `stack` enables a DF traversal of the graph: the next iteration of the search process will pop one of the currently enabled transitions, from the current architecture state, trying to extend the search path down to the `goal`. When a dead end is reached (no transitions are enabled in the current state), the search process implicitly backtracks to a previous graph node by popping from the top of the `stack` a previously pushed context. This enables the complete traversal of the search graph (breadth search). The search process is iterated until the `goal` is reached or there is no more transition to explore (line 17). In this latter case, the requested change is rejected (line 36).

Three examples of evolution plans calculated by our solver are presented in the next sections.

4.2. First experiment: requirement change

The first scenario addresses a requirement change. The initial HAS architecture enables to switch on / off the lights at specific hours (*cf.* Figure 25). However, it does not enable any control on light intensity. To add this new functionality, an architect should modify the HAS specification. This corresponds to a top-down evolution since the change starts at the highest abstraction level. A solution is to replace the *Light* component role by a new one (*Luminosity*) that enables intensity control. Figure 26 presents the initial architecture specification and the evolved one.

An extract of the instantiation of the *Arch_specification* machine corresponding to the HAS is presented in Figure 27.

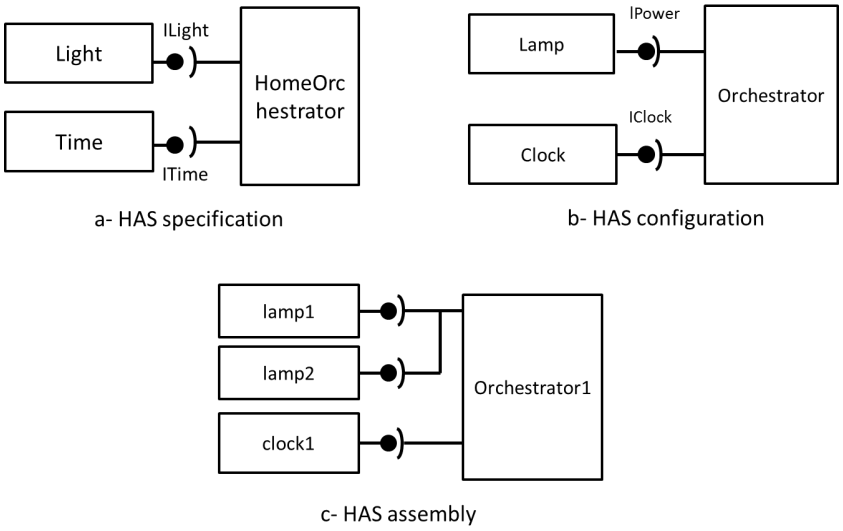


Figure 25: Architecture definitions of the HAS

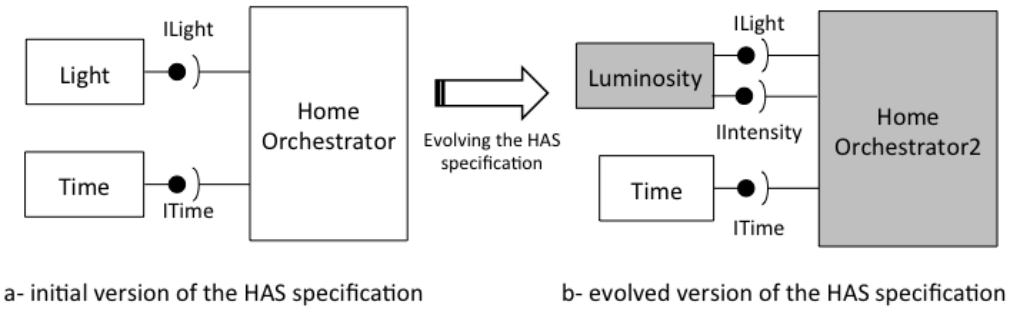


Figure 26: Evolving the HAS specification by role replacement

4.2.1. Evolution goal and initiated change

The initiated change consists in replacing the *Light* component role (*cr1*) by the *Luminosity* component role (*cr1a*). This corresponds to the execution of the role substitution operation on the HAS specification:

| **spec_replaceRole**$(HAS\_spec, cr1, cr1a)$

The following goal is thus given to the solver, based on the post-conditions of the substitution operation, defining the change that must be achieved by the evolution process:

$GOAL == global\_consistency \wedge global\_coherence \wedge cr1a \in spec\_components(HAS\_spec) \wedge$
$cr1 \notin spec\_components(HAS\_spec)$

The solver then calculates an evolution plan that can restore the consistency and coherence of the architecture that may have been altered by the initial change.

compRole $:= \{cr1, cr1a, cr2, cr3, cr3a\}||$
comp_name $:= \{cr1 \mapsto Light, cr1a \mapsto Luminosity, cr2 \mapsto Time,$
$cr3 \mapsto HomeOrchestrator,$
$cr3a \mapsto HomeOrchestrator2\}||$
arch_spec $:= \{HAS\_spec\}||$
spec_components $:= \{HAS\_spec \mapsto \{cr1, cr2, cr3\}\}||$
spec_connections $:= \{HAS\_spec \mapsto \{$
$((cr3, rintILight) \mapsto (cr1, pintILight)),$
$((cr3, rintITime) \mapsto (cr2, pintITime)), \}\}||$
spec_clients $:= \{(HAS\_spec \mapsto \{(cr3, rintILight), (cr3, rintITime)\}\}||$
spec_servers $:= \{(HAS\_spec \mapsto \{(cr1, pintILight), (cr2, pintITime)\}$

Figure 27: Instantiation of the *Arch_specification* machine for the HAS

4.2.2. Triggered change

The intended role substitution entails the addition of a new server interface (the *IIntensity* provided interface) which must be connected to restore the consistency of the HAS specification (all interfaces must be connected at specification level). The solver generates the plan presented in Figure 28 to restore the consistency of the HAS specification.

$spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1a, pintILight))$
$spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime))$
$spec\_deleteRole(HAS\_spec, cr3)$
$spec\_addRole(HAS\_spec, cr3a)$
$spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1a, pintILight2))$
$spec\_connect(HAS\_spec, (cr3a, rintITime2), (cr2, pintITime))$
$spec\_connect(HAS\_spec, (cr3a, rintIIntensity), (cr1a, pintIIntensity))$

Figure 28: HAS specification consistency restoration plan

Change entails the disconnection of all the required interfaces, the deletion of the initial orchestrator (*cr3*), the addition of a new orchestrator (*cr3a*) and finally the connection of all the required interfaces (this is enough to get all the interfaces connected and satisfy the interaction consistency property at specification level).

After consistency is verified for specification, change is propagated to the configuration in order to restore the coherence of the architecture definition.

4.2.3. Change propagation to the configuration

Coherence is altered due to the new requirement defined by the specification. Indeed, the initial HAS configuration (*cf.* Figure 25) does not correctly implement all the roles of the evolved HAS specification. Figure 29 details the instantiation of the *Arch_configuration* machine corresponding to the initial HAS configuration.

$$
\begin{array}{l}
\textbf{compClass} := \{cl1, cl1a, cl2, cl3, cl3a, cl2a\}|| \\
\textbf{comp\_name} := \{cl1 \mapsto Lamp, cl1a \mapsto AdjustableLamp, cl2 \mapsto Clock, \\
\quad cl3 \mapsto Orchestrator, cl3a \mapsto AndroidOrchestrator, \\
\quad cl2a \mapsto AndroidClock\}|| \\
\textbf{configuration} := \{HAS\_config\}|| \\
\textbf{config\_components} := \{HAS\_config \mapsto \{cl1, cl2, cl3\}\} \\
\textbf{config\_connections} := \{HAS\_config \mapsto \{ \\
\quad ((cl3, rintIPower) \mapsto (cl1, pintIPower)), \\
\quad ((cl3, rintIClock) \mapsto (cl2, pintIClock))\}
\end{array}
$$

Figure 29: Initial HAS configuration in Formal Dedal

Change propagation is therefore needed to restore coherence. The restoration plan found by the solver is presented in Figure 30.

$$
\begin{array}{l}
config\_replaceClass(HAS\_config, cl1, cl1a) \\
config\_disconnect(HAS\_config, (cl3, rintILamp), (cl1, pintILamp)) \\
config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock)) \\
config\_deleteClass(HAS\_config, cl3) \\
config\_addClass(HAS\_config, cl3a) \\
config\_connect(HAS\_config, (cl3a, rintILamp2), (cl1a, pintILamp2)) \\
config\_connect(HAS\_config, (cl3a, rintIClock2), (cl2, pintIClock)) \\
config\_connect(HAS\_config, (cl3a, rintIIntensity), (cl1a, pintIIntensity))
\end{array}
$$

Figure 30: Coherence restoration plan for the HAS configuration

It first consists in replacing the *Lamp* component class by the *AdjustableLamp* component class. This operation does not require any modification of the connections, as it is based on the substitution principle between the two component classes (the *AdjustableLamp* class is a specialization of the *Lamp* class). The situation is different regarding the *Orchestrator* component class. It cannot be simply replaced by the existing *AndroidOrchestrator* component class, which is a valid implementation of the *HomeOrchestrator2* role. Indeed, as it holds an extra required interface, the *AndroidOrchestrator* component class is not a specialization of the *Orchestrator* component class. Nonetheless, the solver is able to find a suitable plan to restore consistency

in this more difficult situation. The *Orchestrator* component class is disconnected and removed. The *AndroidOrchestrator* component class is then added and connected. This way, the configuration is consistent (all required interfaces are connected and the configuration is composed of a unique connected graph of components) and coherent with the specification (every role is implemented in the configuration).

4.2.4. Change propagation to the assembly

After coherence is reached in the configuration, change is propagated to the architecture assembly. Here again, coherence is altered because the current HAS assembly is not a valid instantiation of the evolved HAS configuration. Figure 31 details the initial state of the corresponding *Arch_assembly* machine.

$$
\begin{array}{l}
\textbf{compInstance} := \{ci11, ci12, ci1a1, ci1a2, ci2, ci2a, ci3, ci3a\}|| \\
comp\_instantiates := \{ci11 \mapsto cl1, ci12 \mapsto cl1, ci1a1 \mapsto cl1a \\
\quad ci1a2 \mapsto cl1a, ci2 \mapsto cl2, ci2a \mapsto cl2 \\
\quad ci3 \mapsto cl3, ci3a \mapsto cl3\}|| \\
\textbf{compInstance\_name} := \{ci11 \mapsto lamp1, ci12 \mapsto lamp2, ci1a1 \mapsto adjustableLamp1, \\
\quad ci1a2 \mapsto adjustableLamp2, ci2 \mapsto clock1 \\
\quad ci3 \mapsto orchestrator1, ci3a \mapsto androidOrchestrator1, \\
\quad ci2a \mapsto androidClock1\}|| \\
\textbf{assembly} := \{HAS\_assembly\}|| \\
\textbf{assm\_components} := \{HAS\_assembly \mapsto \{ci11, ci12, ci2, ci3\}\} \\
\textbf{assm\_connections} := \{HAS\_assembly \mapsto \{ \\
\quad ((ci3, rintIPowerInst) \mapsto (ci11, pintIPowerInst)), \\
\quad ((ci3, rintIClock) \mapsto (ci2, pintIClockInst)), \ldots\}
\end{array}
$$

Figure 31: Initial HAS architecture assembly

The coherence restoration plan presented in Figure 32 is generated by the solver to propagate changes. First, the client interfaces of the *Orchestrator* component instance are disconnected. Then, the two *Light* component instances are replaced by *AdjustableLight* component instances (as allowed by the substitution principle). The *Orchestrator* component instance is removed and an *AndroidOrchestrator* component instance is added. As explained for the configuration coherence restoration, substitution is not possible because of the extra required interfaces of the *AndroidOrchestrator* component. Fortunately, an evolution plan can still be found so that every component class in the configuration is instantiated at least once in the assembly. Finally, all the required interfaces are connected to compatible provided interfaces, maintaining a consistent assembly.

$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))$
$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci2, pintILampInst2))$
$assm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci12, pintIClockInst))$
$assm\_replaceInstance(HAS\_assembly, ci1, ci1a1)$
$assm\_replaceInstance(HAS\_assembly, ci12, ci1a2)$
$assm\_removeInstance(HAS\_assembly, ci3)$
$assm\_deployInstance(HAS\_assembly, ci3a)$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a1, pintILampInst1))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a1, pintIIntensityInst1))$
$assm\_bind(HAS\_assembly, (ci3a, rintIClockInst), (ci2, pintIClockInst))$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a2, pintILampInst2))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a2, pintIItensityInst2))$

Figure 32: Coherence restoration plan for the HAS architecture assembly

4.3. Second experiment: implementation change

The second scenario addresses an implementation change. The objective is to enable the control of the building through a mobile device (running Android OS for example). To adapt the current implementation to Android, the *Orchestrator* component class (*cl3*) should be removed and replaced with an Android compatible one (*cl3a*). Change is initiated at the configuration level, which entails a mixed evolution: bottom-up because the change has to be propagated to the higher level specification and top-down because it has to be propagated also to the lower assembly level. Figure 33-a shows the initial implementation of the HAS while Figure 33-b shows the evolved one.

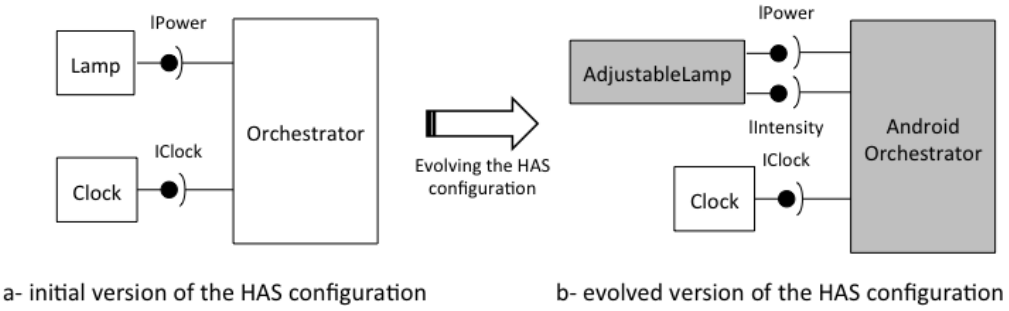


Figure 33: Evolving the HAS configuration by component substitution

4.3.1. Initiated change

Change is initiated by deleting the initial orchestrator (*cl3*) and adding the Android compatible one (*cl3a*). This is processed by the following sequence of operations:

$config\_disconnect(HAS\_config, (cl3, rintILamp), (cl1, pintILamp))$
$config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock))$
$config\_deleteClass(HAS\_config, cl3)$
$config\_addClass(HAS\_config, cl3a)$

To start the evolution process, the following goal is given to the solver:

$GOAL == global\_consistency \wedge global\_coherence \wedge cl3a \in config\_components(HAS\_config) \wedge$
$cl3 \notin config\_components(HAS\_config)$

4.3.2. Triggered change

The generated triggered change is listed in Figure 34. To restore consistency, all component classes must be correctly connected. The *AndroidOrchestrator* component class requires an additional server interface to control the intensity of light. The *Lamp* component class (*cl1*) is suitably replaced with *AdjustableLamp* (*cl1a*) that provides the *IIntensity* server interface. This is another illustration of the solving capabilities of our approach.

$config\_connect(HAS\_config, (cl3a, rintIClock2), (cl2, pintIClock))$
$config\_replaceClass(HAS\_config, cl1, cl1a)$
$config\_connect(HAS\_config, (cl3a, rintIPower2), (cl1a, pintIPower2))$
$config\_connect(HAS\_config, (cl3a, rintIIntensity), (cl1a, pintIIntensity))$

Figure 34: HAS configuration consistency restoration plan

After configuration consistency is verified, change is propagated to the architecture specification.

4.3.3. Change propagation to the specification

The current HAS specification is not any more a good design model of the new version of the HAS configuration. This corresponds to an erosion problem as light intensity control is not included in the current specification. Hence, a new specification version is required to keep architecture descriptions coherent.

Change is propagated to the HAS specification (*cf.* Figure 35) by replacing the *HomeOrchestrator* role (*cr3*) with the *HomeOrchestrator2* (*cr3a*). To do so, the *HomeOrchestrator* role is disconnected and deleted. Then the *HomeOrchestrator2* role is added. On the other way, the *Luminosity* role (*cr1a*) can be directly substituted for the *Light* role (*cr1*). This enforces coherence between the specification and the configuration. Finally, the connection of all client interfaces is sufficient to restore the consistency of the specification (no pending interfaces; a unique connected component graph).

$spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1, pintILight))$
$spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime))$
$spec\_deleteRole(HAS\_spec, cr3)$
$spec\_addRole(HAS\_spec, cr3a)$
$spec\_replaceRole(HAS\_spec, cr1, cr1a)$
$spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1a, pintILight2))$
$spec\_connect(HAS\_spec, (cr3a, rintIIntensity), (cr1a, pintIIntensity))$
$spec\_connect(HAS\_spec, (cr3a, rintITime2), (cr2, pintITime))$

Figure 35: HAS specification coherence restoration plan

4.3.4. Change propagation to the assembly

The current version of the HAS assembly is no more a valid instantiation of the evolved HAS configuration. Change has to be propagated at assembly level to restore coherence (*cf.* Figure 36).

$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))$
$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci12, pintILampInst2))$
$assm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci2, pintIClockInst))$
$assm\_removeInstance(HAS\_assembly, ci3)$
$assm\_deployInstance(HAS\_assembly, ci3a, cl3a)$
$assm\_replaceInstance(HAS\_assembly, ci11, ci1a1)$
$assm\_replaceInstance(HAS\_assembly, ci12, ci1a2)$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a1, pintILampInst1a))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a1, pintIItensityInst1))$
$assm\_bind(HAS\_assembly, (ci3a, rintIClockInst), (ci2, pintIClockInst))$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a2, pintILampInst2a))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a2, pintIItensityInst2))$

Figure 36: HAS assembly coherence restoration plan

In a similar way to specification coherence restoration, the *Orchestrator* component instance (*ci3*) is disconnected and deleted. An *AndroidOrchestrator* component instance (*ci3a*) is added to the assembly. Two *AdjustableLight* component instances (*ci1a1*) and (*ci1a2*) are substituted for the existing *Light* component instances (*ci11*) and (*ci12*). This restores the coherence of the assembly with the configuration. The server interfaces of the components are then bound to compatible provided interfaces, so that the assembly remains consistent (no pending server interfaces; a unique connected component graph).

4.4. Third experiment: runtime change

The third scenario addresses a runtime change. It corresponds to a bottom-up evolution since the change is initiated at the lowest abstraction

level. Because of a dry battery, the clock device in the building is out of service. This environmental change induces the dysfunction of the *clock1* driver (*ci2*). The objective is to find a solution to dynamically repair the architecture in order to maintain the functionalities of the system.

Figure 37 shows the initial and evolved version of the HAS assembly.

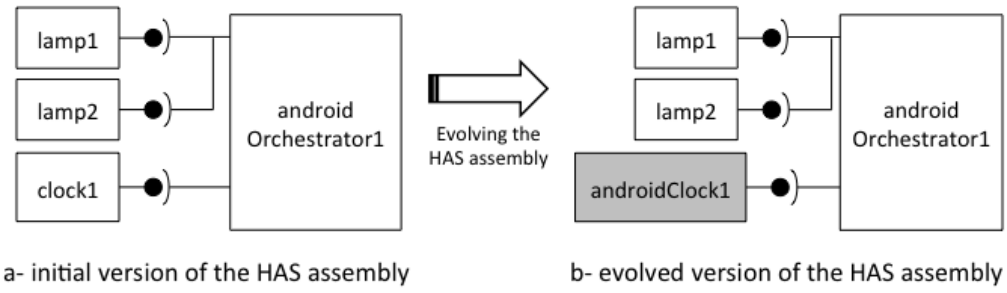


Figure 37: Evolving the HAS assembly by component instance substitution

4.4.1. Initiated change

clock1 (*ci2*) must be replaced by another component instance that provides the same services. An instance of the *AndroidClock* component class, *androidClock1* (*ci2a*), is thus chosen to replace *clock1*. The initiated change is handled by the following operations:

$$| \; replaceInstance(HAS\_assembly, ci2, ci2a)$$

The solver then searches an evolution plan that reaches the following goal:

$$| \; GOAL == global\_consistency \wedge global\_coherence \wedge ci2a \in assm\_components(HAS\_assembly) \wedge ci2 \notin assm\_components(HAS\_assm)$$

4.4.2. Triggered change

The component instance replacement does not alter the consistency of the assembly architecture. However, coherence with the configuration architecture has to be reestablished. Indeed, the evolved assembly architecture is not a valid instantiation of the current configuration architecture since the *ci2a* component instance does not instantiate the *cl2* component class.

4.4.3. Change propagation to the configuration

Change propagation induces the substitution of the *AndroidClock* component class (*cl2a*) for the *Clock* component class (*cl2*), which amounts to the following evolution plan:

$| \ replaceClass(HAS\_config, cl2, cl2a)$

As connections are preserved by the substitution operation, the consistency of the configuration is also preserved. The evolution plan thus includes no other operation.

4.4.4. Change propagation to the specification

The component class substitution preserves the coherence between the specification and the configuration. Indeed, when a component class implements a given role, any component subclass, as a substitute, also implements the role. As a consequence, no change needs to be propagated to the specification.

4.5. Performance evaluation

The performance of the solver has been measured during the three experiments, in order to evaluate the influence of our proposed heuristics. Tests were run on a standard PC (2.5 GHZ Intel Core i5, 8 GB SDRAM) under Windows 7. Test of the three evolution scenarios are then performed first using *DF* and then using *DF* enhanced with heuristics (*H-DF*) to compare the results. Table 1 shows the average time in milliseconds of 5 runs for each evolution scenario, using depth-first search without heuristics (*DF*) and with heuristics (*H-DF*).

	Change level	**DF (ms)**	**H-DF (ms)**
Exp 1	specLevel (initial)	3260	2100
	configLevel	3254	1393
	asmLevel	26738	1926
Exp 2	configLevel (initial)	4712	2537
	specLevel	8733	1896
	asmLevel	TIME-OUT	1927
Exp 3	asmLevel (initial)	4747	1184
	configLevel	TIME-OUT	2351
	specLevel (not affected)	–	–

Table 1: Performance evaluation

Timeout is set to 3 minutes. Results doubtlessly show the benefits of a custom solver that integrates specific heuristics. The order and number of

evolution rules may differ from a generated evolution plan to another (our algorithms are not deterministic as they make random choices when sets of equivalent elements are considered, such as a set of candidate main artifacts) but all generated plans are valid and lead to the same goal state.

A more precise performance evaluation, based on a larger set of experiments and a theoretical study of the combinatorial complexity of the search space is needed. Performance is indeed an inherent limitation for search-based software engineering, as the resolution time of solvers generally grows exponentially depending on the size of the problems. Designing and integrating new heuristics to cut down resolution time is promising (we can for instance preferentially choose transitions that generate no or little incoherence in the architecture model).

5. Related work

This section presents three areas of related work. The first area is that of software architecture evolution which is the main theme of this work. It presents a survey of the main state-of-the-art evolution approaches our work can be compared to. The second area is that of formal modeling languages. It presents a brief comparison of seven formal modeling languages including B. The third area describes other approaches based on model transformation and integration of semi-formal and formal methods. These approaches do not necessarily focus on architecture evolution but they present interesting alternatives from the technical point of view.

5.1. Software architecture evolution

Most of the approaches dealing with architecture evolution adopt an ADL to model architectures and propose a mapping between the ADL and a runtime framework in order to implement the change and enable dynamic evolution. C2-SADEL [24], Darwin [25], ArchWare [26] and Plastik [27] fall into this category. C2-SADEL models architectures in the C2 style [28] and provides multiple component subtyping mechanisms to favor reuse and enable architecture evolution. Its tool support is Dradel, an environment that enables the mapping between architectural description and the implementation by translating them into Java code. The tool supports static evolution by applying changes on architectural descriptions first and then implementing them. The architecture analysis however is limited since no powerful analysis techniques were integrated. Darwin and ArchWare (which provides π-

ADL [29] as an ADL) focus on modeling dynamic structures. They both rely on π-calculus to define the semantics of architecture constructs and guarantee a reliable interaction between components and compile architecture descriptions into code. ArchWare also proposes π-ARL [30] an architecture refinement language to evolve architecture descriptions by stepwise refinement. Plastik was also proposed to deal with dynamic reconfigurations. It relies on Armani, an extension of the ACME [31] ADL to enable invariants expression and reconfigurations properties. Compared to the previous approaches, Plastik has the advantage to map its ADL to OpenCOM [32], a runtime component model dedicated to component-based programming and proposing built-in reconfiguration operations. The main shortcoming of these approaches is that they don't consider changes as first-class elements and focus more on how to implement architecture evolution rather than specify, analyze and propagate it. Moreover, adopted ADLs hardly cover the entire CBSD process. The specification level (necessary to guide reuse) and assembly level (that describes the software at runtime) are often missing. Finally, the coherence between architectural descriptions and implementation is not guaranteed since evolution is processed top-down only.

Recent work by Sanchez *et al.* [33] proposes an architecture-based re-engineering approach to evolve and maintain legacy software. The principle is to produce a high level architecture description of the legacy system so that it becomes easy to reason about change and then reversely use the produced knowledge to modify source code. The approach is guided through a bidirectional transformation and relies on Archery [34], an ADL for modeling architecture patterns corresponding to translated code parts. Targetted at legacy system re-engineering, this work is different from our proposal on the evolution of component-based software systems developed by a reuse-based process.

Other recent approaches show a particular interest to specifying architecture evolution as first-class entities. A first example is the work of Tamazalit, Le Goaer *et al.* [35, 36]. The authors introduce the notion of *evolution styles*, first-class entities that can be specified and classified for reuse to evolve a particular family of systems. Evolution styles include evolution operations that can be specialized, composed and instantiated to deal with change. Barnes *et al.* [37] adopt a wider definition of evolution styles and introduce the concept of evolution paths as a way to plan the evolution of domain-specific software systems. A path is an evolution trace leading from an initial architecture to a desired target architecture. An evolution style refers to a family

of evolution paths sharing common properties. It includes operations, constraints and functions to evaluate paths according to quality metrics. Path constraints can be formally specified using the *path constraint language*, a specific extension of LTL (Linear Temporal Logic). While the computability of the language was proved, as far as we know, there is no existing model checker to support the automated analysis of path constraints. The authors also propose a solution [38] to automate evolution planning using PDDL [39] (the Planning Domain Definition Language). However, this approach still lacks automation since no translation from any ADL to PDDL specification was proposed. Moreover, the evolution is specified and planned beforehand. In our approach, changes are not necessarily expected and the architect intervenes only to validate the work of the evolution manager.

Another closely related work is the one of Hansen, Ingstrup and others [40, 41]. The authors propose an approach to model and analyze runtime architectural change. They opt for a runtime architecture model that closely maps to the OSGi [4] platform to facilitate implementation and for Alloy [42] as a relational first-order logic modeling language to formalize the static and dynamic (operations) concepts of the architecture model. The choice of Alloy is motivated by its support for object-oriented modeling and its accompanying analyzer that enables automated verification. The objective is to apply architectural changes without violating some predefined properties. For this purpose, the authors model the reconfiguration planning as a predicate satisfaction problem with pre- and post-conditions. Then, they run the Alloy SAT solver to find sequences of the model instances satisfying the problem where the first instance satisfies the pre-conditions and the last instance satisfies the post-conditions. This work is similar to ours in the sense that both aim to provide a reliable and automated way to handle architectural changes. It proposes an interesting alternative for resolving evolution using the constraint-solving technique. However, this work focuses only on one level of change which is runtime. Moreover, the formalized architecture model is dependent on OSGi. Finally, the work lacks automation, since no automatic translation from ADL models to Alloy models was proposed.

[4] http://www.osgi.org/Main/HomePage

5.2. Comparison of formal modeling languages

Formal modeling brings abstraction, precision and rigor to software systems. It intervenes at the very early stages of software development to give a formal specification of system requirements. Resulting models constitute unambiguous descriptions that enable software analysis, verification and validation. Several languages and methods were proposed to aid formal modeling. Formal languages provide abstractions to represent concepts, properties over them and possibly behavior. However, they differ in expressiveness, underlying semantics and purpose. Some languages focus more on descriptions and how to make formal modeling more accessible whereas others focus more on automated analysis neglecting expressiveness. A good formal language must be a compromise between both aspects. In the following, we compare seven formal modeling languages. These languages are B [9], Z [43], OCL [44], Alloy [42], VDM [45], Coq [46] and Agda [47].

B, Z and VDM are quite similar in term of expressiveness since they were basically designed for theorem-proving. All of them enable to express properties practically in the same way and support almost the same types (In addition, VDM supports real numbers). However there are some subtle differences between them. Z is more abstract while VDM and B are more low level and intended to be refined into code. Both VDM and B adopt a similar structure that realizes abstract state machines. They explicitly separate the declarative (structure) from the dynamic (operations) part and, unlike Z, they separate pre-conditions from post-conditions. B has the particularity to modify variables by assignments like in programming languages while in VDM and Z, pre and post states must be explicit.

Coq and Agda are proof assistants designed for the verification of functional programs. Unlike the previously mentioned formal modeling languages, Coq and Agda are implementations of type theories rather than set theory. They support higher order logic, polymorphism, dependent types, as well as inductive types. Set theoretic operators (*e.g.* $\cup$, $\cap$), for instance, are not directly predefined in such systems. Unlike B and VDM, these languages do not implement state machines. Therefore, there is no built-in structure that explicitly defines variables, invariants and operations.

OCL and Alloy are different and were designed for different purposes. OCL was basically developed to express constraints that can not be expressed using graphical notations on UML diagrams. It has an object-oriented notation and heavily relies on navigation. Hence predicate expressions are sometimes

verbose comparing to the mathematical notation adopted by the other languages. Alloy is a structural modeling language inspired by Z. It was designed for supporting fully automated analysis. Being strictly first-order, Alloy is less expressive than the other languages [48]. For instance, set of sets and predicates over relations are not directly expressible with Alloy.

Regarding analysis support, all these languages are typed and hence support type-checking. Theorem-proving is supported by Coq, Agda, Z, B and VDM which were basically designed for software correctness. Model-checking and constraint solving is only supported by B, with the ProB tool, and Alloy, with the Alloy analyzer. To some extent, Jaza [49], an animator for Z, enables constraint-solving on small domains. However, Z is limited in terms of model-checking capabilities. This is due to the high abstract nature of the Z language making its handling challenging [50]. Nevertheless, continuous attempts to build a model checker for Z are undertaken [51].

B seems to be the best compromise between expressiveness and analysis support. Alloy could also be a good alternative in our case. However, regardless its expressiveness, it presents another shortcoming. As witnessed in Torlak *et al.* [52], Alloy lacks support of partial instances. Partial instances are explicit representations of instances included in the specification of the model. This is central in our approach since instances are generated automatically from graphical models and injected in B specifications (so-called deep embedding technique [53]). Montaghami *et al.* [54] argued that this feature enables a number of capabilities such as test-driven development, regression testing, modeling by example, and combined modeling and meta-modeling. The authors also proposed a syntax extension of Alloy to support partial instance definition but, as far as we know, this feature is not yet integrated in the last version of Alloy [55].

5.3. Alternative formal approaches

Integration between semi-formal and formal methods is gaining more and more interest in software engineering. On the one hand, semi-formal languages, such as UML [56], offer graphical notations that significantly ease modeling. On the other hand, formal modeling languages provide a strong support for automated software analysis. Several works benefit from combining both kinds of notation to validate their approaches.

Ledru *et al.* [57] propose an approach based on the transformation of UML into B to validate security policies for information systems. They use their

B4MSecure [5] tool to generate B specifications corresponding to a security model. Conjointly, they use ProB to validate security policy scenarios.

Keznikl *et al.* propose the ARCAS method [58], an automated approach to generate connections solutions for middleware architectures. Given a connector specification, the approach translates it into a corresponding Alloy model and performs constraint-solving to find connector instances that realize the specification.

Macedo *et al.* propose Echo [59], an Eclipse-based tool for model repair and transformation using model finding. Given a set of meta-models with internal constraints (specified using OCL) and a set of inter-model consistency rules (specified using QVT-R [60] transformations), Echo can detect inconsistencies on derived models and keep them consistent with their corresponding meta-models and between them as well. The detection and repair mechanism is based on translating MDE [61] artifacts (meta-models and their annotations with OCL and QVT-R rules) to Alloy. The output is then analyzed using a procedure built on top of Alloy solver that generates consistent models as close as possible to the original ones.

6. Conclusions and future work

Managing software architecture evolution throughout the whole software lifecycle is a significant issue. This paper proposes an approach to manage the evolution of component-based software architectures. Thanks to the three-level Dedal architecture model, our approach handles change at three abstraction levels of software architectures: specification, implementation and deployment. The evolution process is driven by an evolution management model that captures changes initiated at any abstraction level, controls their impact to preserve / restore consistency and propagates them to other levels to maintain global coherence.

The proposed evolution management model is based on the B formal language. Using our solver built on top of the ProB tool, it enables the generation of reliable evolution plans as sequences of change operations. The feasibility of our approach is demonstrated by experimenting on three evolution scenarios that each addresses change in a different abstraction level.

The limitation of this work is its scalability. This limitation is classical in comparable works as architecture descriptions can be considered as graphs

[5] http://b4msecure.forge.imag.fr/

(of connected software components) the size of which can theoretically be arbitrarily big. Establishing evolution plans therefore amounts to exploring all possible change action combinations on these graphs to restore properties that can be seen as (local or global) constraints on these graphs. Scalability issue is an inherent limitation for search-based software engineering problems. However, such limitation is mitigated by two factors. First, architecture descriptions are often limited in size as architects prefer to split them in intelligible parts of moderate size using hierarchical composition, an asset of CBSD [1]. Secondly, instead of using an off-the-shelf agnostic B solver, we proposed our own solver that integrates problem-specific heuristics that decrease the calculation time.

Threats to the validity of our approach lie in the example scenarios that we have considered for experimental validation. Although, the examples cover all kinds of scenarios, experimenting with real architecture descriptions might reveal unforeseen issues (scalability, efficiency of heuristics, *etc.*). Further experiments on real case studies is therefore necessary to fully validate our approach.

As future work, we would like to extend our definition of the consistency property in order to include behavioral consistency as described in Taylor *et al.* [21] and thus cover all their identified five kinds of consistency. This would amount in considering architectural protocols and component behavior.

Another interesting research direction would be to integrate the notion of evolution style [36] in our evolution management model. The idea is to enable the generation of multiple candidate evolution plans that can be evaluated considering non-functional properties (*e.g.* quality, cost, time) as proposed by Barnes *et al.* [37].

Regarding the technical aspect, we are investigating new heuristics to improve the performance of our solver and reduce complexity.

7. Acknowledgements

The authors warmly thank the anonymous reviewers for their in-depth reading of the paper and their helpful comments that made it possible to greatly improve its quality.

References

[1] H. V. Vliet, Software Engineering: Principles and Practice, 3rd Edition, Wiley Publishing, 2008.

[2] H. P. Breivold, I. Crnkovic, M. Larsson, A systematic review of software architecture evolution research, Information and Software Technology 54 (1) (2012) 16 – 40. doi:http://dx.doi.org/10.1016/j.infsof.2011.06.002.
URL http://www.sciencedirect.com/science/article/pii/S0950584911001376

[3] T. Mens, S. Demeyer, Software Evolution, Springer, 2008.

[4] D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.

[5] L. de Silva, D. Balasubramaniam, Controlling Software Architecture Erosion: A Survey, Journal of Systems and Software 85 (1) (2012) 132–151.

[6] H. Y. Zhang, C. Urtado, S. Vauttier, Architecture-centric component-based development needs a three-level ADL, in: Proceedings of the 4th European Conference of Software Architecture, Vol. 6285 of Lecture Notes in Computer Science, Springer, Copenhagen, Denmark, 2010, pp. 295–310.

[7] H. Y. Zhang, L. Zhang, C. Urtado, S. Vauttier, M. Huchard, A three-level component model in component-based software development, in: Proceedings of the 11th of the International Conference on Generative Programming: Concepts and Experiences, ACM, Dresden, Germany, 2012, pp. 70–79.

[8] I. Crnkovic, S. Sentilles, A. Vulgarakis, M. Chaudron, A classification framework for software component models, IEEE Transactions on Software Engineering 37 (5) (2011) 593–615.

[9] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, USA, 1996.

[10] D. Cansell, D. Méry, Foundations of the B method, Computers and Informatics 22 (2003) 1–31.

[11] P. Behm, P. Benoit, A. Faivre, J.-M. Meynadier, Meteor: A Successful Application of B in a Large Project, in: J. Wing, J. Woodcock, J. Davies

(Eds.), FM99 Formal Methods, Vol. 1708 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1999, pp. 369–387.

[12] D. Cansell, G. Gopalakrishnan, M. Jones, D. Mry, A. Weinzoepflen, Incremental proof of the producer/consumer property for the PCI protocol, in: D. Bert, J. Bowen, M. Henson, K. Robinson (Eds.), ZB 2002:Formal Specification and Development in Z and B, Vol. 2272 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2002, pp. 22–41.

[13] Atelier B, ClearSy, Aix-en-Provence (F) http://www.atelierb.eu/ accessed 09/01/2015.

[14] The B-Toolkit User's Manual, B-Core (UK) Limited.

[15] D. Delahaye, C. Dubois, C. March, D. Mentr, The BWare project: Building a Proof Platform for the Automated Verification of B Proof Obligations, in: Y. Ait Ameur, K.-D. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, Vol. 8477 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2014, pp. 290–293.

[16] M. Leuschel, M. Butler, ProB: An Automated Analysis Toolset for the B Method, International Journal on Software Tools for Technology Transfer 10 (2) (2008) 185–203.

[17] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: why reuse is so hard, IEEE Software 12 (6) (1995) 17–26. doi:10.1109/52.469757.

[18] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: Why reuse is still so hard, IEEE Software 26 (4) (2009) 66–69. doi:10.1109/MS.2009.86.

[19] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H. Y. Zhang, Formal rules for reliable component-based architecture evolution, in: Formal Aspects of Component Software - 11th International FACS Symposium revised selected papers, Bertinoro, Italy, 2014, pp. 127–142.

[20] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H. Y. Zhang, An evolution management model for multi-level component-based software architectures, in: The 27th International Conference on Software Engineering

and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015, 2015, pp. 674–679. doi:10.18293/SEKE2015-172.
URL http://dx.doi.org/10.18293/SEKE2015-172

[21] R. Taylor, N. Medvidovic, E. Dashofy, Software architecture: Foundations, Theory, and Practice, Wiley, 2009.

[22] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H. Y. Zhang, Towards automating the coherence verification of multi-level architecture descriptions, in: Proceedings of the 9th ICSEA, Nice, France, 2014, pp. 416–421.

[23] M. Leuschel, J. Bendisposto, Directed model checking for B: An evaluation and new techniques, in: J. Davies, L. Silva, A. Simao (Eds.), Formal Methods: Foundations and Applications, Vol. 6527 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 1–16. doi:10.1007/978-3-642-19829-8_1.

[24] N. Medvidovic, D. S. Rosenblum, R. N. Taylor, A Language and Environment for Architecture-based Software Development and Evolution, in: Proceedings of the 21st ICSE, 1999, pp. 44–53.

[25] J. Magee, J. Kramer, Dynamic structure in software architectures, ACM SIGSOFT Software Engineering Notes 21 (6) (1996) 3–14.

[26] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti, ArchWare: Architecting Evolvable Software, in: F. Oquendo, B. Warboys, R. Morrison (Eds.), Software Architecture, Vol. 3047 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 257–271. doi:10.1007/978-3-540-24769-2_23.
URL http://dx.doi.org/10.1007/978-3-540-24769-2_23

[27] A. Joolia, T. Batista, G. Coulson, A. T. A. Gomes, Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform, in: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE, Washington, USA, 2005, pp. 131–140.

[28] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, A Component- and Message-based Architectural Style

for GUI Software, in: Proceedings of the 17th International Conference on Software Engineering, ICSE '95, ACM, New York, USA, 1995, pp. 295–304.

[29] F. Oquendo, Pi-ADL: An Architecture Description Language Based on the Higher-order Typed Pi-calculus for Specifying Dynamic and Mobile Software Architectures, SIGSOFT Software Engineering Notes 29 (3) (2004) 1–14.

[30] F. Oquendo, Pi-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures, SIGSOFT Software Engineering Notes 29 (5) (2004) 1–20. doi:10.1145/1022494.1022517.
URL http://doi.acm.org/10.1145/1022494.1022517

[31] D. Garlan, R. Monroe, D. Wile, ACME: An Architecture Description Interchange Language, in: Proceedings of Centre for Advanced Studies Conference, IBM Press, 1997, p. 7.

[32] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, A component model for building systems software, in: Software Engineering and Applications (SEA '04), Cambrigde, 2004, pp. 684–689.

[33] A. Sanchez, N. Oliveira, L. S. Barbosa, P. Henriques, A perspective on architectural re-engineering, Science of Computer Programming 98, Part 4 (2015) 764 – 784. doi:http://dx.doi.org/10.1016/j.scico.2014.02.026.
URL http://www.sciencedirect.com/science/article/pii/S0167642314000938

[34] A. Sanchez, L. Barbosa, D. Riesco, Bigraphical Modelling of Architectural Patterns, in: F. Arbab, P. lveczky (Eds.), Formal Aspects of Component Software, Vol. 7253 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 313–330. doi:10.1007/978-3-642-35743-5_19.
URL http://dx.doi.org/10.1007/978-3-642-35743-5_19

[35] D. Tamzalit, M. Oussalah, O. L. Goaer, A. Seriai, Updating Software Architectures : A Style-Based Approach, in: Proceedings of the International Conference on Software Engineering Research and Practice &

Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1, 2006, pp. 336–342.

[36] O. L. Goaer, D. Tamzalit, M. Oussalah, A. Seriai, Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures, in: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland, 2008, pp. 311–318. doi:10.1109/COMPSAC.2008.104.
URL http://dx.doi.org/10.1109/COMPSAC.2008.104

[37] J. M. Barnes, D. Garlan, B. Schmerl, Evolution styles: foundations and models for software architecture evolution, Software and Systems Modeling 13 (2) (2014) 649–678.

[38] J. M. Barnes, A. Pandey, D. Garlan, Automated planning for software architecture evolution, in: Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pp. 213–223. doi:10.1109/ASE.2013.6693081.
URL http://dx.doi.org/10.1109/ASE.2013.6693081

[39] D. McDermott, PDDL-The Planning Domain Definition Language, Tech. rep., Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998).

[40] M. Ingstrup, K. M. Hansen, Modeling architectural change: Architectural scripting and its applications to reconfiguration, in: Joint Working IEEE/IFIP Conference on Software Architecture, WICSA/ECSA, 2009, pp. 337–340. doi:10.1109/WICSA.2009.5290670.

[41] K. M. Hansen, M. Ingstrup, Modeling and Analyzing Architectural Change with Alloy, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 2257–2264. doi:10.1145/1774088.1774560.
URL http://doi.acm.org/10.1145/1774088.1774560

[42] D. Jackson, Alloy: A Lightweight Object Modelling Notation, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 256–290.

[43] J. M. Spivey, The Z Notation: A Reference Manual, Prentice Hall International (UK) Limited, 1992.

[44] OCL, 2.3.1 specification, http://www.omg.org/spec/OCL/2.3.1/ accessed 09/01/2015.

[45] C. B. Jones, Systematic Software Development Using VDM (2nd Ed.), Prentice-Hall, 1990.

[46] Y. Bertot, P. Castran, G. Huet, C. Paulin-Mohring, Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions, Texts in theoretical computer science, Springer, Berlin, New York, 2004, donnes complmentaires http://coq.inria.fr.
URL http://opac.inria.fr/record=b1101046

[47] U. Norell, Dependently Typed Programming in Agda, in: Proceedings of the 6th International Conference on Advanced Functional Programming, AFP'08, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 230–266.
URL http://dl.acm.org/citation.cfm?id=1813347.1813352

[48] D. Jackson, Software Abstractions: Logic, Language, and Analysis, Appendix E: Alternative Approaches, The MIT Press, 2006.

[49] M. Utting, Jaza user manual and tutorial, http://www.cs.waikato.ac.nz/ marku/jaza/ accessed 03/08/2015.

[50] G. Smith, L. Wildman, Model Checking Z Specifications Using SAL, in: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB 2005: Formal Specification and Development in Z and B, Vol. 3455 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 85–103. doi:10.1007/11415787_6.
URL http://dx.doi.org/10.1007/11415787_6

[51] J. Derrick, S. North, A. Simons, Z2SAL: a translation-based model checker for Z, Formal Aspects of Computing 23 (1) (2011) 43–71. doi:10.1007/s00165-009-0126-7.
URL http://dx.doi.org/10.1007/s00165-009-0126-7

[52] E. Torlak, D. Jackson, Kodkod: A Relational Model Finder, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4424 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 632–647. doi:

10.1007/978-3-540-71209-1_49.
URL http://dx.doi.org/10.1007/978-3-540-71209-1_49

[53] J. Svenningsson, E. Axelsson, Combining Deep and Shallow Embedding for EDSL, in: H.-W. Loidl, R. Pea (Eds.), Trends in Functional Programming, Vol. 7829 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 21–36. doi:10.1007/978-3-642-40447-4_2.
URL http://dx.doi.org/10.1007/978-3-642-40447-4_2

[54] V. Montaghami, D. Rayside, Extending alloy with partial instances, in: J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, E. Riccobene (Eds.), Abstract State Machines, Alloy, B, VDM, and Z, Vol. 7316 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 122–135. doi:10.1007/978-3-642-30885-7_9.
URL http://dx.doi.org/10.1007/978-3-642-30885-7_9

[55] Alloy lanaguge reference, http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf accessed 25/07/2015.

[56] UML, 2.5 specification, http://www.omg.org/spec/UML/2.5/Beta2/ accessed 09/01/2015.

[57] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, M.-A. Labiadh, Taking into Account Functional Models in the Validation of IS Security Policies, in: C. Salinesi, O. Pastor (Eds.), Advanced Information Systems Engineering Workshops, Vol. 83 of Lecture Notes in Business Information Processing, Springer, Berlin, Heidelberg, 2011, pp. 592–606.

[58] J. Keznikl, T. Bure, F. Plil, P. Hntynka, Automated resolution of connector architectures using constraint solving (ARCAS method), Software & Systems Modeling 13 (2) (2014) 843–872. doi:10.1007/s10270-012-0274-8.
URL http://dx.doi.org/10.1007/s10270-012-0274-8

[59] N. Macedo, T. Guimaraes, A. Cunha, Model repair and transformation with Echo, in: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013, pp. 694–697. doi:10.1109/ASE.2013.6693135.

[60] OMG, MOF 2.0 Query/View/Transformation (QVT), version 1.1, http://www.omg.org/spec/QVT/1.1/ accessed 24/07/2015.

[61] D. C. Schmidt, Guest Editor's Introduction: Model-Driven Engineering, Computer 39 (2) (2006) 25–31.