Information and Software Technology xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis

Gabriela Arévalo [a,f,*], Stéphane Ducasse [c], Silvia Gordillo [b,e], Oscar Nierstrasz [d]

[a] FI – Universidad Austral, Avda. Juan de Garay 125, AR-1063, Buenos Aires, Argentina
[b] LIFIA – UNLP, 50 y 115, AR-1900, La Plata, Argentina
[c] INRIA Lille – Nord Europe- ADAM Team, Parc Scientifique de la Haute Borne, 40 Avenue Halley, FR-59650, Villeneuve d'Ascq, France
[d] SCG – IAM, Universitaet Bern, Neubrückstrasse 10, CH-3012, Bern, Switzerland
[e] CICPBA, 526 between 10 and 11, AR-1900, La Plata, Argentina
[f] CONICET, Av. Rivadavia 1917, AR-1033, Buenos Aires, Argentina

ARTICLE INFO

Article history:
Received 30 December 2008
Received in revised form 15 April 2010
Accepted 21 May 2010
Available online xxxx

Keywords:
Object-oriented development
Class hierarchy schemas
Source code analysis
Formal Concept Analysis

ABSTRACT

Context: Inheritance is the cornerstone of object-oriented development, supporting conceptual modeling, subtype polymorphism and software reuse. But inheritance can be used in subtle ways that make complex systems hard to understand and extend, due to the presence of implicit dependencies in the inheritance hierarchy.
Objective: Although these dependencies often specify well-known *schemas* (i.e., recurrent design or coding patterns, such as hook and template methods), new unanticipated dependency schemas arise in practice, and can consequently be hard to recognize and detect. Thus, a developer making changes or extensions to an object-oriented system needs to understand these implicit contracts defined by the dependencies between a class and its subclasses, or risk that seemingly innocuous changes break them.
Method: To tackle this problem, we have developed an approach based on Formal Concept Analysis. Our Formal Concept Analysis based-Reverse Engineering methodology (FoCARE) identifies undocumented hierarchical dependencies in a hierarchy by taking into account the existing structure and behavior of classes and subclasses.
Results: We validate our approach by applying it to a large and non-trivial case study, yielding a catalog of *hierarchy schemas*, each one composed of a set of dependencies over methods and attributes in a class hierarchy. We show how the discovered dependency schemas can be used not only to identify good design practices, but also to expose bad smells in design, thereby helping developers in initial reengineering phases to develop a first mental model of a system. Although some of the identified schemas are already documented in existing literature, with our approach based on Formal Concept Analysis (FCA), we are also able to identify previously unidentified schemas.
Conclusions: FCA is an effective tool because it is an ideal classification mining tool to identify commonalities between software artifacts, and usually these commonalities reveal known and unknown characteristics of the software artifacts. We also show that once a catalog of useful schemas stabilizes after several runs of FoCARE, the added cost of FCA is no longer needed.

1. Introduction

Inheritance is the cornerstone of object-oriented development, supporting conceptual modeling, subtype polymorphism and software reuse [75]. There is often a tension between these three uses of inheritance. During analysis, inheritance is used to model specialization of domain concepts; during design, generic components are specified by exploiting polymorphism; and during implementation, inheritance is often used as a pure reuse mechanism to define new classes from existing ones [29,46].

The different uses of inheritance in large class hierarchies result in *explicit* and *implicit dependencies* within such class hierarchies. *Explicit* dependencies are clearly identifiable from source code, such as the definitions of subclasses. On the other hand, *implicit* dependencies are induced by uses of *self* and *super* sends, method cancellations in subclasses, method redefinitions in subclasses, or state access from subclasses.

In general, inheritance is not a simple incremental definition mechanism, but it usually works in strong interaction with other

* Corresponding author. Tel./fax: +54 11 5921 8000.
E-mail addresses: garevalo@austral.edu.ar (G. Arévalo), ducasse@inria.fr (S. Ducasse), gordillo@sol.info.unlp.edu.ar (S. Gordillo), oscar@iam.unibe.ch (O. Nierstrasz).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.05.010

object-oriented mechanisms [75], such as overriding, method cancellations and encapsulation.

Inheritance combined with these object-oriented mechanisms can make class hierarchies difficult to understand, because unexpected and implicit behavior based-dependencies can appear. Here are some illustrations of the possible difficulties a reengineer may face when he has to modify an existing and often unknown hierarchy.

- *Template Method in superclasses*. When defining a class, the developer has to understand if the superclass imposes restrictions regarding the behavior of subclasses. This is the case of *Template Method*, which defines a skeleton of an algorithm in a method in a class, deferring the implementation of the algorithm steps to its subclasses [2,13,65]. For example, Fig. 1a shows *Template Method* between the superclass **KeyedCollection** and the subclasses **RBSmallDictionary** and **MethodDictionary**. The method **atIfAbsentPut** calls the methods **atIfAbsent** and **atPut** which are abstract in the class itself, but concrete in the subclasses. This is a classical use of subtyping. This restriction is implicit and the developer can only detect the presence of this design pattern by attentively reading the code. Although modern development environments are capable of automatically generating stub methods for subclasses of abstract classes, the general problem of detecting and enforcing implicit constraints between classes and subclasses is unsolved.
- *Inheritance for implementation reuse*. The use of inheritance for implementation reuse (subclassing) can violate subclassing contracts from structural and behavioral viewpoints. For example, in Fig. 1b the class **Dictionary** is a subclass of the class **Set**. **Dictionary** elements can only be removed by their key values, so the method **removeIfAbsent** inherited from class **Set** is overridden to raise an error. From a structural viewpoint inheritance is used for mere implementation purposes, and it forces the developer to cancel inherited methods or to override superclass behavior with error messages. From the behavioral viewpoint, they are incompatible because this hierarchy does not respect the Liskov Substitution Principle [52].
- *Yo-yo problem*. The behavior of a class is specified by its methods and the reuse of its superclass behavior via *self* and *super* sends. The fact that *self* is dynamic (i.e., *self* sends are resolved only at runtime to the appropriate receiver method) while *super* is static (i.e., methods are looked up statically in the superclass of class containing the method issuing the super call), can make it difficult to understand the run-time behavior of a given class [77,27]. When a class hierarchy is deep, the flow is implicit and is difficult to follow since the programmer has to keep flipping between different class definition. This phenomenon is known as the *yoyo effect* [74].
- *Inappropriate state usage*. Object-oriented languages support various visibility mechanisms for instance variables defined in classes. In languages where instance variables are "protected" by default, such as Smalltalk, it is not possible to prevent subclasses from inappropriately accessing inherited state. While a class often hides its state from its clients, supporting a good encapsulation of internal representation, subclasses can directly access superclass state creating unexpected dependencies. Classes such as **OrderedCollection** (in Fig. 1c) do not expose their state to client classes, since there are no accessors. However, this does not prevent subclasses from accessing directly (without accessors) the state defined by the class. Subclasses can therefore violate the encapsulation of the class. A class should ideally act as a provider of (inherited) services for its subclasses [71]. For example, Fig. 1c shows that class **SortedCollection** accesses the attribute **lastIndex** of the class **OrderedCollection** without using accessors. This happens because the attributes in **OrderedCollection** are used internally in the class itself, and they are not public to its clients. The only way that the class **SortedCollection** can access superclass state is by violating the encapsulation of the superclass.

During initial maintenance phases, when the developer needs to develop a first mental model of an object-oriented system, he should understand the contracts implicitly defined by the dependencies between a class and its subclasses to avoid introducing changes that break these contracts, or to eventually fix existing problems before modifying the system [19,72].

A key issue for our analysis is that the contracts can be defined by a *simple* dependency, such as the reuse of behavior *via a self call* from one subclass and engaging its superclass, or by a combination of certain dependencies, as we have seen in the description of a *Template Method*. The contract defined with a *simple* dependency can be easily detected using a query-based tool, because we know exactly how the contract is defined [78]. But the contracts defined by a combination of (explicit and implicit) dependencies cannot easily be detected like that, because we do not know *a priori* which are the contracts, how they are defined in terms of sets of dependencies between classes and if there are relationships between them. Without Formal Concept Analysis (FCA), we would need to generate all the possible combinations of (explicit and implicit) dependencies between classes, resulting an exponential number of candidate contracts. Each combination can define an unexpected contract, which should be identified and verified if it appears in the class hierarchy. Thus, the process can be expensive in terms of performance and processing, and FCA is a better choice in our context.

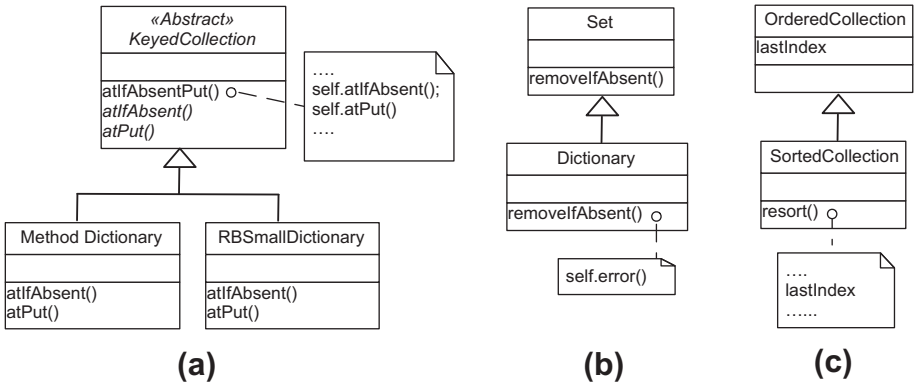

Fig. 1. Schemas in class hierarchies: (a) Template Method. (b) Cancelled Behavior. (c) Access to superclass state.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

We have developed an approach called Formal Concept Analysis based-Reverse Engineering (FoCARE) to detect dependencies in a class hierarchy using FCA [4]. While our original approach focused purely on the behavior of classes, the work presented in this paper also takes state into account. FoCARE adopts a pipeline architecture with a set of processing steps that transforms source code in terms of useful high-level entities for reengineering purposes. In our specific case, we identify hierarchy schemas to obtain a first mental map for initial reengineering phases. One of the main features of this methodology is that it is language independent because it uses a model of the source code as input instead of the source code itself. Language independence is achieved by using *Moose*[1] reengineering platform, a research vehicle for reverse and reengineering object-oriented software [61] in the methodology. We use FCA to identify recurring, unanticipated and undocumented patterns revealing the way classes use local (or superclass) behavior and state. The patterns revealed by our analysis, called hierarchy schemas, are composed of a set of *dependencies* over methods and attributes in a class hierarchy. These schemas correspond not only to *known contracts* defining *best practices* but also to *hidden contracts*, *bad smells* and questionable or *irregular practices*.

We validate our approach with an extensive case study: the Smalltalk **Collection** hierarchy, which we selected because (i) it is a complex and essential part of the Smalltalk system, (ii) it combines subtyping and subclassing, (iii) it is an industrial quality class hierarchy that has evolved over 20 years, and (iv) it has been studied by other researchers [11,17,35,46].

The case study reveals non-trivial hierarchy schemas in industrial software systems, which we categorize as follows:

- *Classical Schemas* represent common styles that are used to build and extend a class hierarchy,
- *Irregular Schemas* represent questionable practices which can lead to difficulties when understanding the hierarchy,
- "*Bad Smell*" *Schemas* indicate the presence of likely design defects that should be corrected.

The identified hierarchy schemas help us to answer such questions as:

- Which classes define and use (or not) their own state and behavior?
- Which classes use the state defined in their superclasses?
- Which classes use *template and hook methods* and define behavior for their subclasses [65]?
- Which classes reuse or extend (or not) the behavior of their superclasses?
- Which subclasses redefine or cancel the behavior of their superclasses?

Our methodology also helps us to evaluate FCA itself, and answer such questions as:

- How useful is FCA for software analysis?
- Does FCA scale when applied in practice?
- Considering that there is an algorithm involved in the methodology, how does the performance affect the results?

The two main contributions of this paper are:

A catalog of hidden class dependencies. The catalog of hierarchy schemas provides a good basis for identifying which parts of a system are well-designed and which ones need to be fixed. The approach provides two analysis viewpoints: a *global* view of the system which identifies which kinds of dependencies and practices are present, and a *partial* view of how specific classes are related to others in their hierarchy and how that hierarchy can be modified and extended. Although the catalog is built using FCA, once the catalog reaches a fixed point, the engineer can identify the schemas using a simple query engine with the definitions provided by the presented approach.

FCA as a query-engine The FoCARE methodology demonstrates how FCA can be an effective tool to reveal *implicit patterns* in complex software systems. We show the main difference between the use of FCA and a query-based engine. If we knew in advance how the Hierarchy Schemas should be defined, we could use a query-based tool to identify their occurrences, but it is not the case in our approach. We use FCA because we do not know in advance which are the possible schemas occurring in the class hierarchies, and consequently we do not know the combination of dependencies that characterize these schemas. FCA helps us mainly to discover *implicit* and *unanticipated* schemas introduced in a class hierarchy, and that the patterns may or may not correspond to expected practices.

There are two main differences as compared to our previous work [4]. First, in the current approach we include dependencies that consider access to object state, whereas our earlier work considered only behavior. Thus, we yield more concepts, and hence more schemas of interest than when only behavior is considered. Secondly, we categorize schemas into those that represent *good, irregular* and *bad* design decisions in the class hierarchies.

The paper is structured as follows: Section 2 briefly describes Formal Concept Analysis. Section 3 details the methodology FoCARE and Section 4 presents our mapping of object-oriented dependencies in class hierarchies using FoCARE, and the catalog of hierarchy schemas. Section 5 provides an overview of the results obtained by applying our approach to the Smalltalk **Collection** hierarchy and by showing how **SortedCollection** fits into this hierarchy. Section 6 analyzes several issues related to the application of the FoCARE approach to class hierarchies and the threats to validity of the developed approach. Section 7 compares our approach to related work. Finally, Section 8 concludes and outlines future work.

2. Formal Concept Analysis (FCA) in a nutshell

Formal Concept Analysis (FCA) [30] is a branch of lattice theory that allows us to identify meaningful groupings of *elements* that have common *properties*.[2]

Let us consider a small example about animals and how we categorize them using FCA. The elements are a group of animals *Lion*, *Finch*, *Eagle*, *Hare* and *Ostrich*; and the properties are *Preying*, *Flying*, *Bird* and *Mammal*. Table 1 shows us an *incidence table* indicating which animal has which properties.

Based on these sets, a *context* is defined as a triple $C = (E, P, R)$, where E and P are finite sets of elements and properties respectively, and R is a binary relation between E and P represented in the incidence table. In the example, the elements are the animals, the properties are its features, and the binary relation *is a* is defined by Table 1. For example, the tuple *(Lion, Preying)* is in R, but *(Finch, Mammal)* is not.

Let $X \subseteq E$, $Y \subseteq P$, and $X' = \{p \in P | \ \forall e \in X : (e,p) \in R\}$ and $Y' = \{e \in E | \forall p \in Y : (e,p) \in R\}$, then X' gives us all the *common proper-*

[1] http://www.moosetechnology.org.

[2] In standard FCA literature these are referred to, respectively, as *objects* and *attributes*. We use the terms *element* and *property* to avoid the unfortunate clash with object-oriented terminology.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

Table 1
Incidence table of the animals example.

	Preying	Flying	Bird	Mammal
Lion	X			X
Finch		X	X	
Eagle	X	X	X	
Hare				X
Ostrich			X	

ties of the elements contained in X, and Y' gives us the *common elements* of the properties contained in Y, *e.g.*, $\{Lion,Eagle\}' = \{Preying\}$.

A *concept* is a pair of sets – a set of elements (the *extent*) and a set of properties (the *intent*) (X,Y) – such that $Y = X'$ and $X = Y'$. In other words, a concept is a maximal collection of elements sharing common properties. In Table 1, a concept is a maximal rectangle (up to permutation of rows and columns of the table) we can obtain with relations between animals and its features. For example, *(*{*Finch, Eagle*}*, *{*Flying, Bird*}*)* is a concept, whereas *(*{*Lion*}*, *{*Mammal*}*)* is not, since $\{Lion\}' = \{Mammal\}$, but $\{Mammal\}' = \{Lion,Hare\}$.

The set of all the concepts of a given context forms a *complete partial order* shown in Fig. 2a. We define that a concept (X_0,Y_0) is a *sub-concept* of concept (X_1,Y_1), denoted by $(X_0,Y_0) \leqslant (X_1,Y_1)$, if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$). Inversely we define that the concept (X_1,Y_1) is a *super-concept* of concept (X_0,Y_0). For example, the concept ({*Eagle*},{*Preying*,*Flying*, *Bird*}) is a sub-concept of the concept ({*Eagle*,*Lion*}, {*Preying*}). The set of concepts constitutes a *concept lattice* and there are several algorithms for computing the concepts and the concept lattice for a given context.

There are two alternatives for labelling the concepts. Given a concept $c = (E,P)$, the label $l(c)$ can be defined as:

- $l(c) = (extent(c), intent(c)) = (E,P)$. This means that we label concepts with all the elements and properties calculated for the concept. Fig. 2a shows the lattice of the example with this notation. For example, the concept c_5 has {*Eagle, Finch*} as extent and {*Bird, Flying*} as intent.
- $l(c) = (E_n,P_m)$, if c is the *largest* concept (w.r.t. $\leqslant$) with $p(\in P_m)$ in its intent, and c is the *smallest* concept (w.r.t $\leqslant$) with $e(\in E_n)$ in its extent. The reading rule is as follows: An element e has a property p if and only if there is an upwards leading path from the circle named e to the circle named p. Hence, *Lion* has exactly the properties *Mammal* and *Preying*. We can easily read the extent (and the intent) of each concept by collecting all elements below (respectively all properties above) the given concept. Fig. 2b shows the lattice of the example with this notation. For example, the concept c_5 has {*Finch*} as reduced extent and {*Flying*} as reduced intent.

From the conceptual viewpoint, each concept represents a category of elements described by a set of properties. In the animals example, c_3 represents the mammals, c_4, the preying ones and c_6, the birds. They correspond to *natural* concepts in our restrictive domain of animals. We will use FCA in a similar way to identify recurring *concepts* by analyzing software artifacts and the dependencies between them.

3. FoCARE: architecture of the implementation

The approach to detect hierarchy schemas presented in this paper is based on the FoCARE methodology. FoCARE is language independent and its goal is to use FCA to build tools that identify recurring sets of dependencies in the context of object-oriented software reengineering. It conforms to a pipeline architecture in which the analysis is carried out by a sequence of five processing steps with different input and output artifacts, and two iterative phases, and the output of each step provides the input to the next step.

Fig. 3 shows the different parts of FoCARE detailed as follows.

The processing steps are *Model Import*, *FCA Mapping*, *ConAn Engine*, *Post-Filtering*, *Analysis*, and the iterative phases are *Modelling* and *Interpretation*. First, we explain the processing steps as iterative phases work with them. We describe each step listing the goal of each step and indicating *Input* and *Output* models.

- *Model Import:* Our first step is to build a *model* of the application from the source code. For this purpose we use the *Moose* reengineering platform, a collaborative research platform for software reengineering and visualization [61,26,32,36].*Moose* models the software artifacts of the target system as *language independent* source code level entities. This allows entities to be directly queried and interacted with, consequently providing a convenient way to query and navigate the model with meta-descriptions [24]. Software models in *Moose* conform to the FAMIX family of language-independent metamodels for reengineering [20]. FAMIX is customized for various aspects of code representation (static, dynamic, history). FAMIX core, the metamodel we used in this paper, describes the static structure of software systems in terms of packages, classes, method, attribute, variable and their associated relationships (inheritance, access, and invocation).
  Input: Source code of the application to analyze.
  Output: Moose model of the source code.
- *FCA Mapping:* An FCA Context (Elements, Properties, Incidence Table) is built, mapping from Moose model entities to FCA elements and properties.
  Input: Moose model of source code.
  Output: An FCA context (Elements, Properties, Incidence Table).

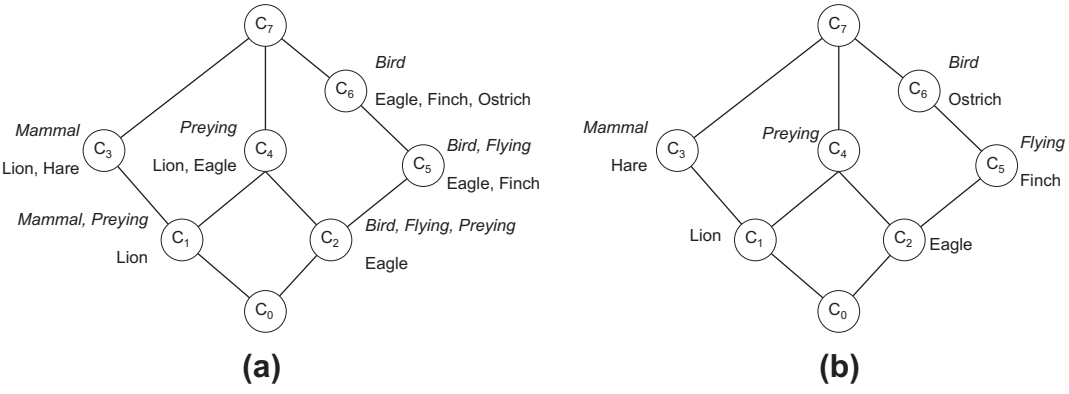

Fig. 2. Lattices of the *Animal* example with (a) complete labelling and (b) reduced labelling.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

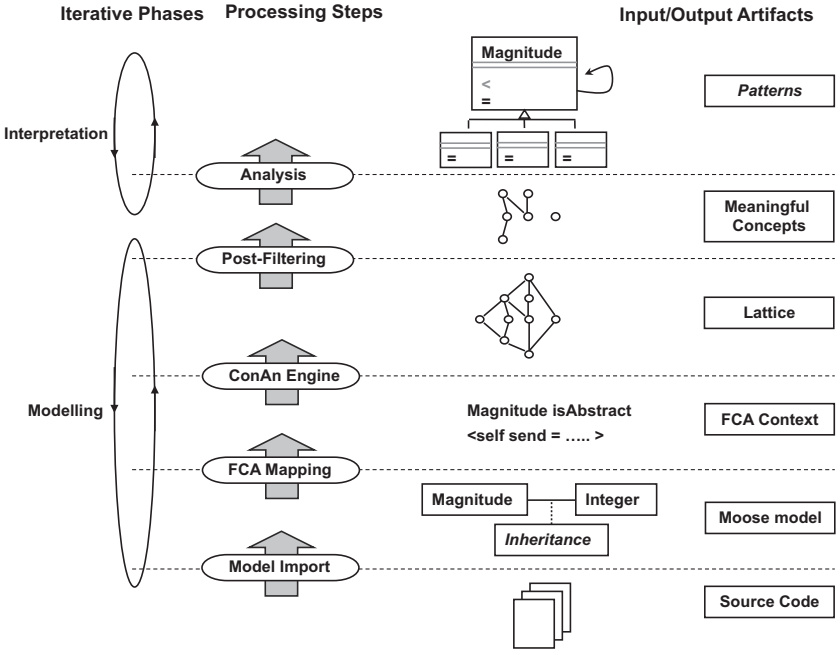

Fig. 3. The overall approach.

- *ConAn Engine:* Once the FCA elements and properties are defined, we run the *ConAn* engine. The *ConAn* engine is a black-box component implemented in VisualWorks 7 which runs the FCA algorithms to build the concepts and the lattice. *ConAn* applies the Ganter algorithm [30] to build the concepts and our own implementation of the algorithm [5] to build the lattice.
  Input: FCA Context (Elements, Properties, Incidence Table).
  Output: Lattice with Concepts.
- *Post-Filtering:* Once the concepts and the lattice are built, each concept constitutes a potential *candidate* for analysis. But not all the concepts are relevant. We have a *post-filtering* process, which is the last step performed by the tool. In this way we filter out meaningless concepts.
  Input: Lattice with Concepts.
  Output: Candidate Concepts.
- *Analysis:* In this step, the software engineer examines the candidate concepts resulting from the previous steps and uses them to explore the different *implicit* dependencies between the software entities and how they determine or affect the behavior of the system.*Input:* Candidate Concepts.*Output: Patterns* (in our specific case, hierarchy schemas).

A key aspect of FoCARE is that the engineer has to iterate over the modeling and the interpretation phases (see Fig. 3) in order to find hierarchy schemas of interest for analyzing a class hierarchy. The *modeling* phase entails a process of experimentation with smaller case studies to find a suitable mapping from the source code model to FCA elements and properties. A particular challenge is to find a mapping that is efficient in terms of identifying meaningful concepts while minimizing the quantity of data to be processed.

In the *interpretation* phase, the output of the modeling phase is analyzed in order to interpret the resulting concepts in the context of the application domain. The useful concepts can then be flagged so that future occurrences can be automatically detected. As more case studies are analyzed, the set of identifiably useful concepts typically increases up to a certain point, and then stabilizes.

In Section 4 we show how we instantiate our architecture to analyze class hierarchies and outline the key issues that must be addressed to use FoCARE in analyzing class hierarchies. *Model Import* is not described in Section 4 because it is performed by MOOSE, a tool that supports automatically this phase in FoCARE. Thus, we only describe the *FCA Mapping*, *ConAn Engine*, *Post-Filtering* and *Analysis*.

4. Analyzing class hierarchies

If we knew which contracts between classes to expect in a given application (and how they are defined), we could use a query-based tool to identify their presence in the software. Unfortunately this is not possible since the range of possible contracts is completely open and therefore we cannot know their definitions (see Section 6). For this reason, we need a technique that detects the presence of implicit patterns in the dependencies between classes.

The analysis of class hierarchies using FCA helps us to identify hierarchy schemas that will show us how the class hierarchies are designed and specifically how classes define their state and methods on their own and in terms of superclasses and subclasses. To detect these schemas, we must model class hierarchies in terms of FCA *elements* and their *properties*. Note that FCA is perfectly neutral in terms of what elements and properties represent. As a consequence we can choose how we model class hierarchies and the contracts (i.e., concepts) we are looking for.

We first describe how we build the corresponding elements and properties (Section 4.1), the incidence table and the lattice (Section 4.2). Then we show which conditions a candidate concept should fulfil to (Section 4.3) and how recurring combinations of properties lead to the dependency schemas of interest (Section 4.4). Fig. 4 shows a part of the *Smalltalk Collection* that we use as an example to show how we analyze a class hierarchy with our approach.

4.1. FCA mapping: elements and properties of classes

Let us analyze our goal illustrating what information we need. We use FCA to identify implicit contracts such as usage of the Template Method Design Pattern. In a Template Method, the elements of interest are methods of a class and its subclasses, and the properties we are interested in are the facts that the template method calls its hook method, that the hook method may be abstract in the

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

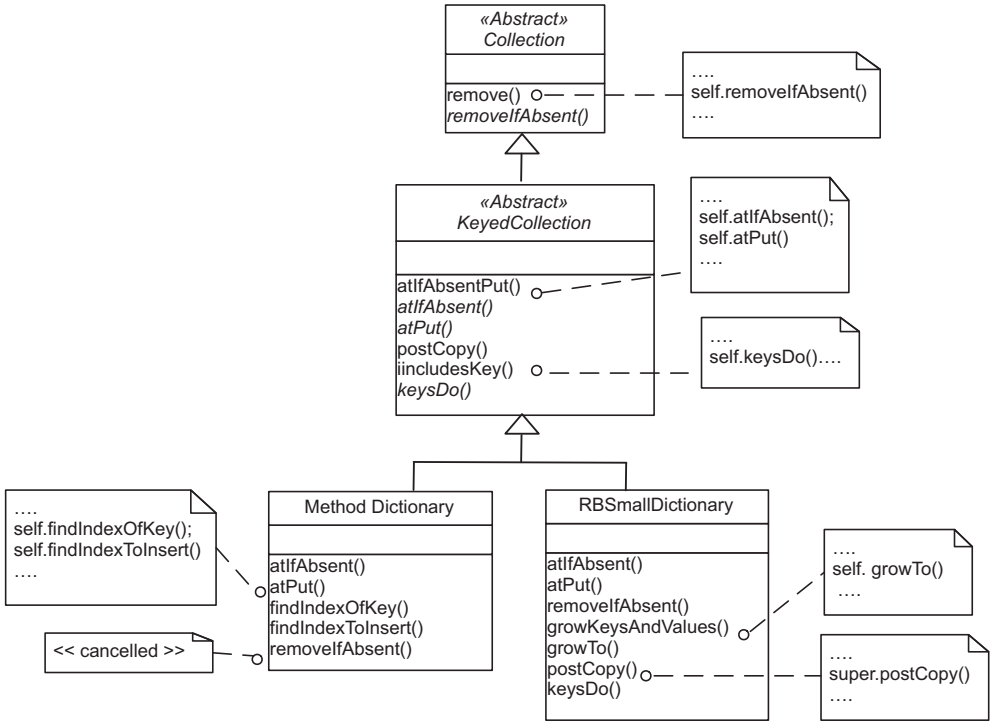

Fig. 4. Example of a class hierarchy.

superclass, and that the hook method is overridden in subclasses. We are also interested if methods access attributes in idiomatic ways, for example, if a subclass accesses an attribute defined in a superclass. In general we do not know what kinds of combinations of dependencies will be present, so we use FCA to detect which contracts are present as concepts, i.e., as sets of elements with similar sets of properties.

4.1.1. FCA elements

Since we want to analyze the inheritance dependencies within a class hierarchy, we build FCA elements using *self* and *super invocations* of methods and *accesses* to the attributes of the classes. We build the FCA elements as pairs of *(Invoking Class C, invoked method m)* and *(Accessing Class C, accessed attribute a)*. The pairs are interpreted as the method *m* is *invoked* in the class *C*, and the attribute *a* is accessed in the class *C* respectively. For example, the element **(Col, removeIfAbsent)** (shown in Table 2) represents that the method **removeIfAbsent** is called in the class **Collection**. We abbreviate the names of the classes in the incidence table and in the lattice (as explained in Section 4.2). The abbreviated names are **Col**, for **Collection**, **KC**, for **KeyedCollection**, **MD**, for **MethodDictionary**, and **RBD**, for **RBSmallDictionary**.

Table 2 shows the elements in our sample class hierarchy (shown in Fig. 4). They are **(Col, removeIfAbsent)**, **(KC, atIfAbsent)**, **(KC, atPut)**, **(MD, findIndexOfKey)**, **(MD, findIndexToInsert)**, **(RBD, growTo)**, **(RBD, postCopy)** and **(KC, keysDo)**.

From FCA representation viewpoint, we do not consider the name of the caller or accessor method as a part of the FCA elements representing invocations or accesses. For example, the FCA element **(Col, removeIfAbsent)** should be (**Collection**, *remove*, **removeIfAbsent**) if we kept the caller method. Instead we just keep **(Col, removeIfAbsent)**. This decision is made to reduce the number of generated FCA elements (hence, noise) in the results. Let us imagine that there is another method *m* that also calls **removeIfAbsent** in **Collection**, then we would have (**Collection**, *remove*, **removeIfAbsent**) and (**Collection**, *m*, **removeIfAbsent**) as FCA elements. The explosion in the number of FCA elements can also generate problems in lattice computation time. As we are interested in the hierarchy dependencies between classes, we just keep one representative in these specific cases. We select **(Col, removeIfAbsent)**, which represents all the calls made to **removeIfAbsent** in the class **Collection**.

Table 2
Incidence relation of class hierarchy example.

	Invoked via self	Invoked via super	Is abstract locally	Is concrete locally	Is concrete in descendant **RBD**	Is concrete in descendant **MD**	Is cancelled in descendant **MD**	Is concrete in ancestor **KC**
(Col, removeIfAbsent)	X		X		X		X	
(KC, atIfAbsent)	X		X		X	X		
(KC, atPut)	X		X		X	X		
(MD, findIndexOfKey)	X			X				
(MD, findIndexToInsert)	X			X				
(RBD, growTo)	X			X				
(RBD, postCopy)		X						X
(KC, keysDo)	X		X		X			

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

4.1.2. FCA properties

In order to define specific properties for each type of element: *invocations* and *accesses*, first we extract some *facts* regarding the code but obtained from Moose model. These facts characterize the dependencies between the class that *defines* a method based on method invocation or an attribute based on its access within a class. Let us see which are the mentioned facts.

Consider the elements (*C*,*m*) and (*C*,*a*) that represent an invocation or an access in class *C*, and the classes C_1 related to *C*, then the facts are:

Applied to attribute and accesses:
- *C accesses a via accessors.*
- *C accesses a without accessors.*
- *C defines a.*

Applied to methods and invocations:
- *C invokes m via self.*
- *C invokes m via super.*
- *C delegates m via super* (This case is a specific case of the *invokes via super*. It happens when the called message has the same name as the invoker method, in this case *m*).
- *m is abstract* in *C*.
- *m is concrete* in *C*.
- *m is cancelled*[3] in *C*.

Applied to both types of elements:
- C_1 *is ancestor of C*.
- C_1 *is descendant of C*.

Let us enumerate some facts that we identify in the example of Fig. 4:

- **RBSmallDictionary** *invokes* **growTo**,
- **postCopy** *is concrete* in **RBSmallDictionary**,
- **atPut** *is abstract* in **KeyedCollection**,
- **removeIfAbsent** *is cancelled* in **MethodDictionary**,
- **Collection** *is ancestor of* **KeyedCollection**,
- **KeyedCollection** *is ancestor of* **RBSmallDictionary**,
- **MethodDictionary** *is descendant of* **KeyedCollection**,
- **RBSmallDictionary** *invokes* **postCopy** *via super*, and
- **KeyedCollection** *invokes* **atPut** *via self*.

We directly adopt some of these facts as properties, and we also combine some of them to obtain the final set of FCA properties. When building FCA elements, we restrict our analysis to *self* and *super* invocations of methods because we want to analyze inheritance dependencies within a class hierarchy. Therefore, the following facts are amongst those that we adopt directly:

- *C invokes m via self.*
- *C invokes m via super.*
- *C delegates m via super.*

For example, Fig. 4 shows that **KeyedCollection** *invokes* **atPut** *via self* and that **RBSmallDictionary** *invokes* **postCopy** *via super*.

The remaining facts are combined to generate the following FCA properties. The left part of the assertion corresponds to FCA property and the right part corresponds to the definition based on the combination of the facts. For example, the property *m is concrete in ancestor* C_1 *of C* is defined by the combination of the facts *C invokes m*, C_1 *defines m*, *m is concrete* in C_1 and C_1 *is ancestor of C*.

- *C accesses local state* (*via accessors* or *without accessors*) = *C accesses a* and *C defines a.*
- *C accesses state in Ancestor* C_1 (*via accessors* or *without accessors*) = *C accesses a*, C_1 *is ancestor of C* and C_1 *defines a.*
- *m is concrete locally* = *C invokes m* and *m is concrete* in *C* (Two more properties can be built using *is abstract* or *is cancelled* instead of *is concrete*).
- *m is concrete in ancestor* C_1 *of C* = *C invokes m*, *m is concrete* in C_1 and C_1 *is ancestor of C* (Two more properties can be built using *is abstract* or *is cancelled* instead of *is concrete*).
- *m is concrete in descendant* C_1 *of C* = *C invokes m*, *m is concrete* in C_1 and C_1 *is descendant of C* (Two more properties can be built using *is abstract* or *is cancelled* instead of *is concrete*).

We have 14 FCA properties in total in our approach.
Let us consider the properties identified in the example of Fig. 4.

- **removeIfAbsent** *is cancelled in descendant* **MethodDictionary** *of* **Collection** = **Collection** *invokes* **removeIfAbsent**, **removeIfAbsent** *is cancelled* in **MethodDictionary** and **MethodDictionary** *is descendant of* **Collection**,
- **postCopy** *is concrete in ancestor* **KeyedCollection** *of* **RBSmallDictionary** = **RBSmallDictionary** *invokes* **postCopy**, **postCopy** *is concrete* in **KeyedCollection** and **KeyedCollection** *is ancestor of* **RBSmallDictionary**, and
- **atPut** *is abstract locally* = **KeyedCollection** *invokes* **atPut** and **atPut** *is abstract* in **KeyedCollection**.

From the FCA representation viewpoint, the format of the some properties will change compared to the format used in our previous explanation. When building the incidence table, the main idea is to check if the element *e* fulfils (or not) a property *p*, such as *p(e)*.

- The properties involving *self*, *super* and *delegates* sends are formulated similarly to *invoked via self*, *invoked via super* and *delegated via super* as presented previously.
- The property involving local state or behavior (for example, *accesses local state*) will not denote any explicit class because in our interpretation the class in the property is the same as mentioned in the element. For example, **atPut** *is abstract locally* means that the element (**KeyedCollection**, **atPut**) fulfils the property *is abstract locally* if **KeyedCollection** *invokes* **atPut**, **KeyedCollection** *defines* **atPut** and **atPut** *is abstract* in **KeyedCollection**.
- However, when the property involves an ancestor or descendant class, the property is slightly different. For example, if we see the expression **postCopy** *is concrete in ancestor* **KeyedCollection** *of* **RBSmallDictionary**, we observe that the class **RBSmallDictionary** of the element **(RBD, postCopy)** seems to be part of the property, but in fact the expression mixes the information regarding the element **(RBD, postCopy)** and the property *is concrete in ancestor* **KeyedCollection**. Thus, when the property involves an ancestor or a descendant related to an element (*C*,*m*) or (*C*,*a*), the property keeps the ancestor or descendant classname.

Thus, the properties in our sample example (shown in Table 2) are *invoked via self*, *invoked via super*, *is abstract locally*, *is concrete locally*, *is concrete in descendant***RBD**, *is concrete in descendant***MD**, *is cancelled in descendant***MD** and *is concrete in ancestor***KC**. The first four properties do not have an explicit class, and the other three involve an ancestor or descendant class of the elements.

[3] In Smalltalk it is possible to "cancel" a method inherited from a superclass by causing it to raise an exception, effectively undefining it. In languages where this is not possible, this property can simply be ignored.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

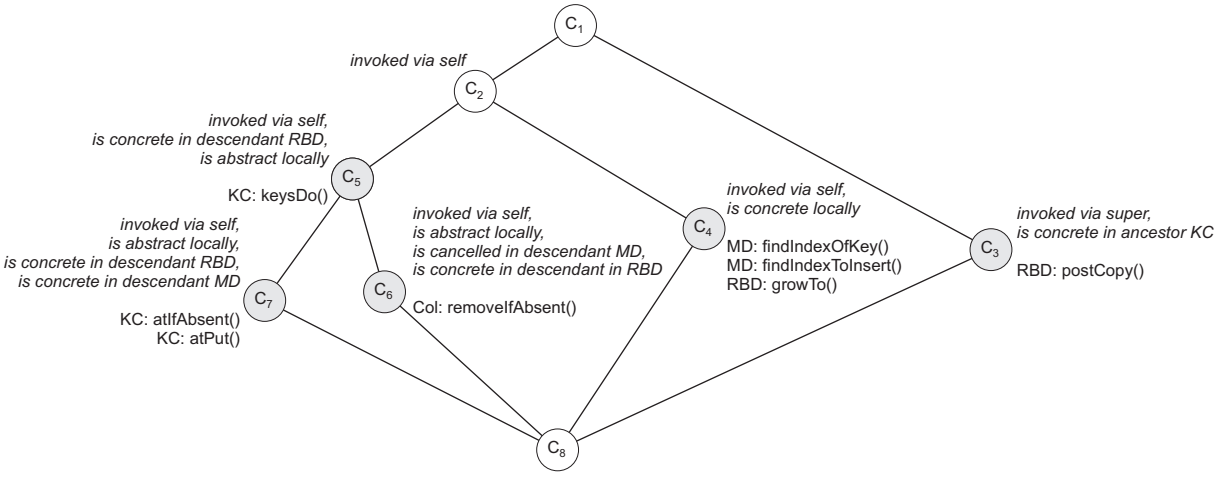

Fig. 5. Lattice calculated from the incidence table (shown in Table 2) of class hierarchy example.

4.2. Incidence table and lattice construction

Once the elements and properties are chosen, we build the corresponding incidence table (shown in Table 2) and calculate the lattice (shown in Fig. 5).

In our specific approach, the FCA elements are *self* and *super* invocations of methods and accesses to attributes of the classes in a class hierarchy expressed as pairs of *(Invoking Class C, invoked method m)* and *(Accessing Class C, accessed attribute a)*. And the FCA properties identify dependencies between the class that *defines* a method based on method invocation or an attribute based on its access within a class.

Regarding the elements, as explained previously, we just keep one representative of each call/access in a class (the first one that we found when analyzing the subject system). This means that if we have several calls to the method *m* or accesses to the attribute *a* in several methods of the class *C*, we just keep one representative of *m* or *a* related to the class *C*. This way, we avoid repeated information in the incidence table, and reduce the amount of data to deal with in the lattice.

Regarding the properties, if they do not have a corresponding invoked method or an accessed attribute that fulfils them, we do not use them in the incidence table. In our case, we do not have the property *is cancelled locally* because there is no invoked method in one class that has a cancelled definition in the same class. Neither we have, for example, the property *is concrete in ancestor* **Collection**, because there is no invoked method in one subclass of **Collection** that has the respective concrete definition in **Collection**.

From the viewpoint of FCA, we ease the lattice understanding using a *combined labelling* of the concepts in the lattice. We use a complete label in the intent, and a reduced label in the extent of the concepts, because we are interested in identifying the set of most restrictive properties that characterizes each invocation or each access. For example, Fig. 6 shows the concepts c_5, c_6 and c_7 with *complete* and *reduced* labelling. If we used the *complete* label in the extent of the concepts and we analyzed c_5, we could say a *candidate schema* (described by c_5) contains **(Col, removeIfAbsent)**, **(KC, atIfAbsent)**, **(KC, atPut)** and **(KC, keysDo)** because they fulfil the set of properties {*invoked via self*, *is concrete in descendant***RBD**, *is abstract locally*}. But in fact, the elements **(Col, removeIfAbsent)** and **(KC, atIfAbsent)** are also contained in the *candidate schema* (described by c_7), and **(KC, atPut)** is contained in one described by c_6. We need every element to be an instance of only one *schema* with most of the properties that characterizes it. The *reduced* labelling of the extent reveals this specific information. On the other hand, if we kept the *reduced* labelling in the intent, we would have that navigate through all the parent chain of concepts to recover the information. For example, identifying the properties of concept c_6 with *reduced* labelling implies to recover also the properties of c_5. As we are interested in the complete definition of *candidate* schema, the *complete* labelling reveals this information.

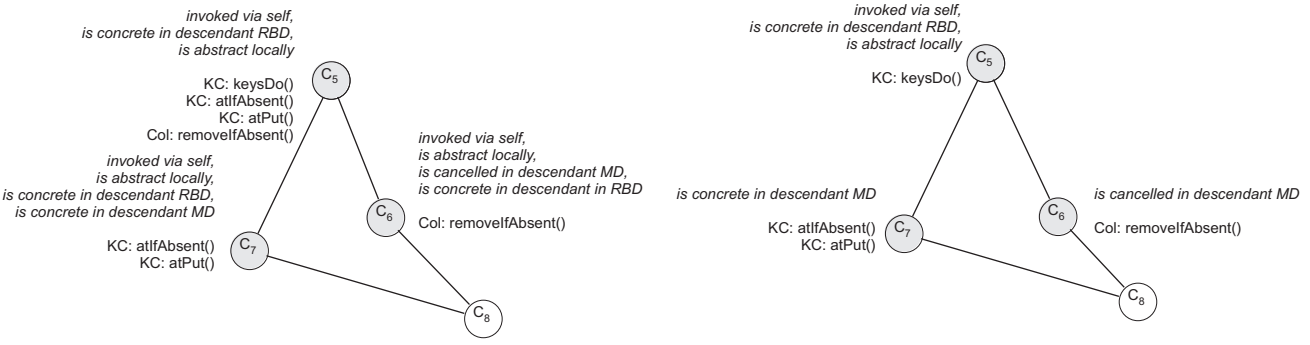

Fig. 6. Complete and reduced labelling of concepts c_5, c_6 and c_7.

With this *combined labelling*, we make sure that an invocation or an access is involved in only one candidate schema (that means that only one is contained in one *Hierarchy Schema*), and that we contain the definition of each *potential* schema in the concept itself.

4.3. Identifying meaningful concepts

Once the lattice is computed, we apply specific filtering criteria to identify the *candidate* concepts for describing hierarchy schemas. The applied criteria comprises the following rules:

- Regarding the concepts, we keep only those concepts with a non-empty extent in the combined labelling. This means that if an invocation is contained in more than one concept, we will just keep the concept that identifies the most restrictive schema.
- Regarding the invocations in a class hierarchy, and the dependencies between the different classes, all the meaningful concepts must show information about at least one of the three dependencies: *local*, *ancestor* and *descendant*, and have at least one property of the set {*invoked via self*, *invoked via super*, *delegated via super*}. Thus, the minimal information shown by a concept is how the methods are invoked and where they are implemented. Most schemas show several properties of the first set {local, ancestor, descendant}, generating interesting schemas that show how different classes in the hierarchy are related.

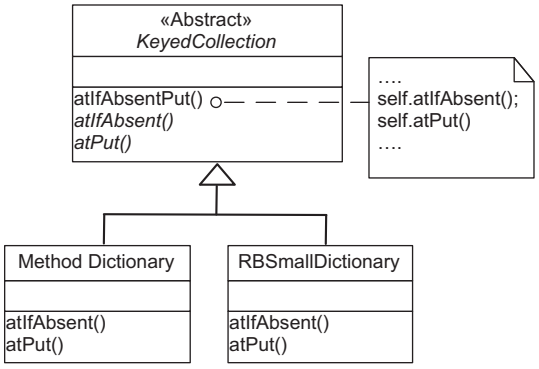

Fig. 7. Template method represented in concept c_7 of the class hierarchy example.

- Regarding the accesses to attributes in a class hierarchy, the meaningful concepts must show at least one of the following dependencies: {accesses local state without accessors, accesses local state with accessors, accesses state without accessors in ancestor }, and can show some dependencies of the complete set of {local, ancestor, descendant, invoked via self, invoked via super, delegated via super}.

The rest of the concepts are discarded. Thus, from our concepts in Fig. 5, we keep the concepts c_3, c_4, c_5, c_6, and c_7 which fulfil the conditions described previously, and we discard the concepts c_1, c_2 and c_8.

4.4. Hierarchy schemas

Hierarchy schemas are built using the set of properties of the filtered concepts of the previous step. Let us see two examples

- The set of properties {*invoked via self*, *is abstract locally*, *is concrete in descendant* C_1} which element *(C,m)* fulfils, shows that the class *C* invokes methods *m* via self that is defined as abstract in the class *C*, and that is implemented as concrete ones in the descendant C_1. This concept reveals a good practice using a *Template Method* showing that the superclass defines the behavior skeleton of the subclasses. Fig. 7 shows this schema in our example represented in the concept c_7.
- The set of properties {*invoked via self*, *is abstract locally*, *is concrete in descendant* C_1, *m is cancelled in descendant* C_2} which element *(C,m)* fulfils, shows that the class *C* defines a protocol that subclasses should implement, but the subclass C_2 cancels partially the inherited protocol. This set shows a bad coding practice, because it is based on an inappropriate use of subclassing. Fig. 8 shows this schema in our example represented in the concept c_6.

Based on the interpretation of the set of properties contained in the concepts, we have named each *Hierarchy Schema* and we have categorized these into three groups: *Classical*, *Irregular* and *"Bad Smell"*. In the following we describe each category and we detail each specific schema with a description and the set of dependencies that define them as concepts in the lattice. In some cases, we observe that several sets of dependencies can define the same schema. Note that we use the notation C_n or C_n (n: 1...m) to indicate the (ancestor or descendant) class(es) related to the invoker

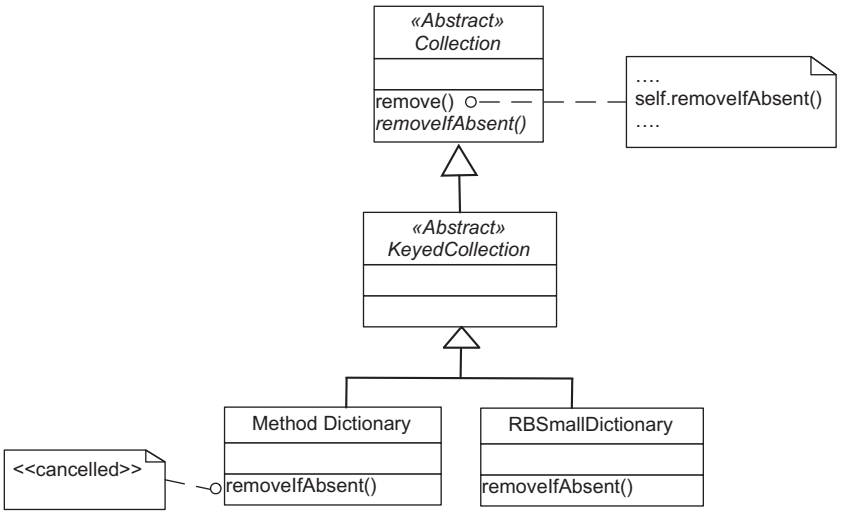

Fig. 8. Cancelled local or inherited behavior represented in concept c_6 of the class hierarchy example.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

class of the invocations or accesses. Unless stated explicitly, ancestor or descendant classes are not necessary direct ones.

- *Classical* Schemas represent *common* idioms/styles that are used to build and extend a class hierarchy, i.e., *best practices* identified by existing literature [8,43,67].

Classical	
Name	Description
Local Direct State Access	Identifies methods that directly access instance variables. *Variations:* using or not using accessors. Properties: *(1) accesses local state without accessors, (2) accesses local state with accessors.*
Local Behavior	Identifies methods defined and used in the class that are not overridden in subclasses. Often represent internal class behavior. Properties: *invoked via self, is concrete locally*
Template and hook	Identifies methods that define template and hook methods. *Variations:* default hooks are abstract or represent a default behavior. Properties: *invoked via self, is abstract locally, is concrete in descendant* C_n.
Redefined Concrete Behavior	Identifies concrete inherited methods that are redefined in the class or in the subclasses. Properties: *invoked via self, is concrete locally, is concrete in ancestor* C_m*, (2) invoked via self, is concrete in descendant* C_n*, is concrete in ancestor* C_m.
Extended Concrete Behavior	Identifies concrete inherited methods that are extended in the class (only *super* send). Properties: *delegated via super, is concrete locally, is concrete in ancestor* C_n.
Reuse of Superclass/State Behavior	Identifies concrete methods that invoke superclass methods by *self* or *super* sends. *Variation:* Extended Concrete Behavior. Properties: *(1) invoked via self, is concrete in ancestor* C_n*, (2) invoked via super, is concrete in ancestor* C_n*, (3) accesses state with accessors in* C_n *(n: 1 ..m).*
Local Behavior overridden in Subclasses	Identifies methods that are overridden in subclasses. Properties: *invoked via self, is concrete locally, is concrete in descendant* C_n.

- *"Bad Smell"* Schemas represent doubtful designs decisions used to build a hierarchy. They are frequently a sign that some parts should be completely changed or even rewritten from scratch.

Bad smells	
Name	Description
Ancestor Direct State Access	Identifies methods that directly access the instance variable of an ancestor, bypassing any accessors. This schema is the consequence of bad use of Reuse of Superclass/State Behavior regarding the ancestor state. Properties: *accesses state without accessors in ancestor* C_n *(n: 1 ..m)*
Cancelled Local Behavior but Superclass Reuse	Identifies concrete inherited methods whose behavior is *cancelled* in the class but whose corresponding superclass behavior is invoked i.e., via a *super* send from a different method. This work around is a common sign of difficulty improperly factoring out common behavior. Properties: *invoked via super, is cancelled locally, is concrete in ancestor* C_n *(n: 1 ..m)*
Abstracting Concrete Methods	Identifies abstract methods overriding concrete ones. Properties: (1) *invoked via self, is concrete locally, is abstract in descendant* C_n *(n: 1 ..m)* (2) *invoked via super, is abstract locally, is concrete in ancestor* C_n *(n: 1 ..m)*
Cancelled local or inherited behavior	Identifies concrete local or inherited methods that are invoked i.e., via *self* send in a class or its superclasses, but are cancelled in subclasses. Method cancellation is a sign of inheritance for code reuse without regard for subtyping. Properties: (1) *invoked via self, is cancelled locally, is concrete in ancestor* C_n, (2) *invoked via self, is concrete locally, is cancelled in descendant* C_n
Broken super send Chain	Identifies methods that are extended (i.e., via a *super* send) at some point in the hierarchy, but are then simply overridden lower in the hierarchy. This can be the sign of a broken subclassing contract. Properties: *delegated via super, is concrete locally, is concrete in ancestor* C_n.

- *Irregular* Schemas represent *irregular* situations used to build the hierarchy. Often the implementation can be improved using minimal changes. They are less serious than *"Bad Smell"* schemas.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

Irregularities	
Name	Description
Inherited and Local Invocations	Identifies methods that are invoked by both *self* and *super* sends within the same class. This may be a problem if the super sends are invoked from a method with a different name. Properties: *(1) invoked via self, invoked via super, is concrete locally, is concrete in ancestor C_n, (2) invoked via self, invoked via super, delegated via super, is concrete locally, is concrete in ancestor C_n.*
Unused Local Behavior but Superclass Reuse	Identifies concrete inherited methods whose behavior is *overridden* but unused in the class, and whose corresponding superclass behavior is invoked *i.e.*, via a *super* send from a different method. Properties: *invoked via super, is concrete locally, is concrete in ancestor C_n.*
Accessor Redefinition	Identifies methods that are accessors in a class but are redefined in the subclass as non-accessor methods. Properties: *invoked via self, is concrete locally, accesses state with accessors in ancestor C_n.*

Remarks. From the FCA viewpoint, it is important to remark that:

- We can have *Equivalent Concepts.* Although each concept is a candidate to be a schema, several concepts represent the same schema. This happens in two different situations. The first one is that one schema can have equivalent concepts if we have several concepts with the same set of properties, except that the superclass and subclasses names of the properties change. For example, the concept c_7 represents an instance of the schema *Template Method*, described by the set of properties *invoked via self*, *is abstract locally*, *is concrete in descendant* **RBD** and *is concrete in descendant* **MD**. If we do not consider the classes that are the parameters of the properties, we obtain the set {invoked via self, is abstract locally, is concrete in descendant} that describes the mentioned schema. In this way, any concept that contains the same set of properties but not with the same parameters, is considered an *equivalent concept* of c_7. The second situation is that one schema can have different set of properties as its definition. For example, the schema Cancelled Local or inherited behavior can be defined with two following intents: (1) *invoked via self, is cancelled locally, is concrete in ancestor C_n*, (2) *invoked via self, is concrete locally*, *is cancelled in descendant C_n*. Both definitions are valid, because they represent the same schema where the method cancellation varies if it is implemented locally or in a descendant.
- A concept with the intent containing the properties *is concrete locally*, *is cancelled locally* or *is abstract locally*, and without those ones that describe a dependency with other superclass(es) or subclass(es) (for example, *is concrete in descendant***RBD**) can represent a schema that is present in several classes, though the classes in the extent of that concept have no relation to each other. For example, the concept c_4 represents the schema Local Behavior, but this schema appears in the classes **MethodDictionary** and **RBSmallDictionary** (according to our example). **MethodDictionary** and **RBSmallDictionary** are not related based on the properties of the concept. This happens because the methods **findIndexOfKey** and **findIndexToInsert** are defined in **MethodDictionary** and the method **growTo** is defined in **RBSmallDictionary**, and they do not have a property in common in terms of a superclass or subclass of **MethodDictionary** or **RBSmallDictionary** respectively. This happens because when using *is concrete|abstract|cancelled locally*, these properties do not specify any explicit class in their parameters, because the relation of the property with the element is implicit.

FoCARE's architecture is implemented as a tool to make it easier to analyze the results regarding a class hierarchy. Fig. 9 shows a screenshot of the tool, in which we see the Hierarchy schemas in the left pane. Clicking on a particular instance of a *schema* will cause its classes to be displayed in the right pane.

5. Detected hierarchy schemas in the Collection hierarchy

We present here the results of our analysis of the Smalltalk Collection hierarchy. This hierarchy is especially interesting because (i) it is part of the core of Smalltalk system, (ii) it makes heavy use of subclassing as well as subtyping, (iii) it is an industrial quality class hierarchy that has evolved over more than 20 years, and (iv) has been studied by other researchers [35,17,11,46]. It has also influenced the design of current C++ and Java collection hierarchies. In VisualWorks, the Smalltalk Collection hierarchy is composed of 104 classes distributed over 8 levels of inheritance. There are 2162 defined methods in all the classes, with 3117 invocations of these methods within the hierarchy and 1146 accesses to the state of the classes defined in the hierarchy.

This case study shows that the approach can effectively identify non-trival schemas in professional software system. We first provide a *global* overview of the identified schemas, and then we focus on the role of the class *SortedCollection* within the collection. We chose this class because it is a good example where several schemas are overlapped in the same class, and this fact helps the developer to understand how it was built and how it works.

5.1. Global view on Collection hierarchy

Analyzing Collection hierarchy, we discovered 451 instances of 16 different identified hierarchy schemas. Table 3 shows the number of detected instances of each schema.

The descriptions of hierarchy schemas provided in Section 4.4 are generic. In what follows we provide detailed analysis of specific instances of some hierarchy schemas, that we consider the most interesting ones.

Classical: Local Direct State Access. This schema identifies classes that define and use their own state directly (using or not the accessors). In the Collection hierarchy, there are 55 classes contained in this schema. Most of the classes are leaves of the hierarchy. This shows that the hierarchy is built using subclassing, since each class extends inherited behavior from the superclasses and provides specific functionality of its own. Only the subhierarchies starting from *String* and *WeakDictionary* have no leaf classes that match this pattern, meaning that eventually these classes either use state of the superclasses or only extend the behavior of the superclasses without extending the state of the superclasses. This schema helps us identify which parts of the hierarchy have *behavior oriented* or *state oriented* classes [7].

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

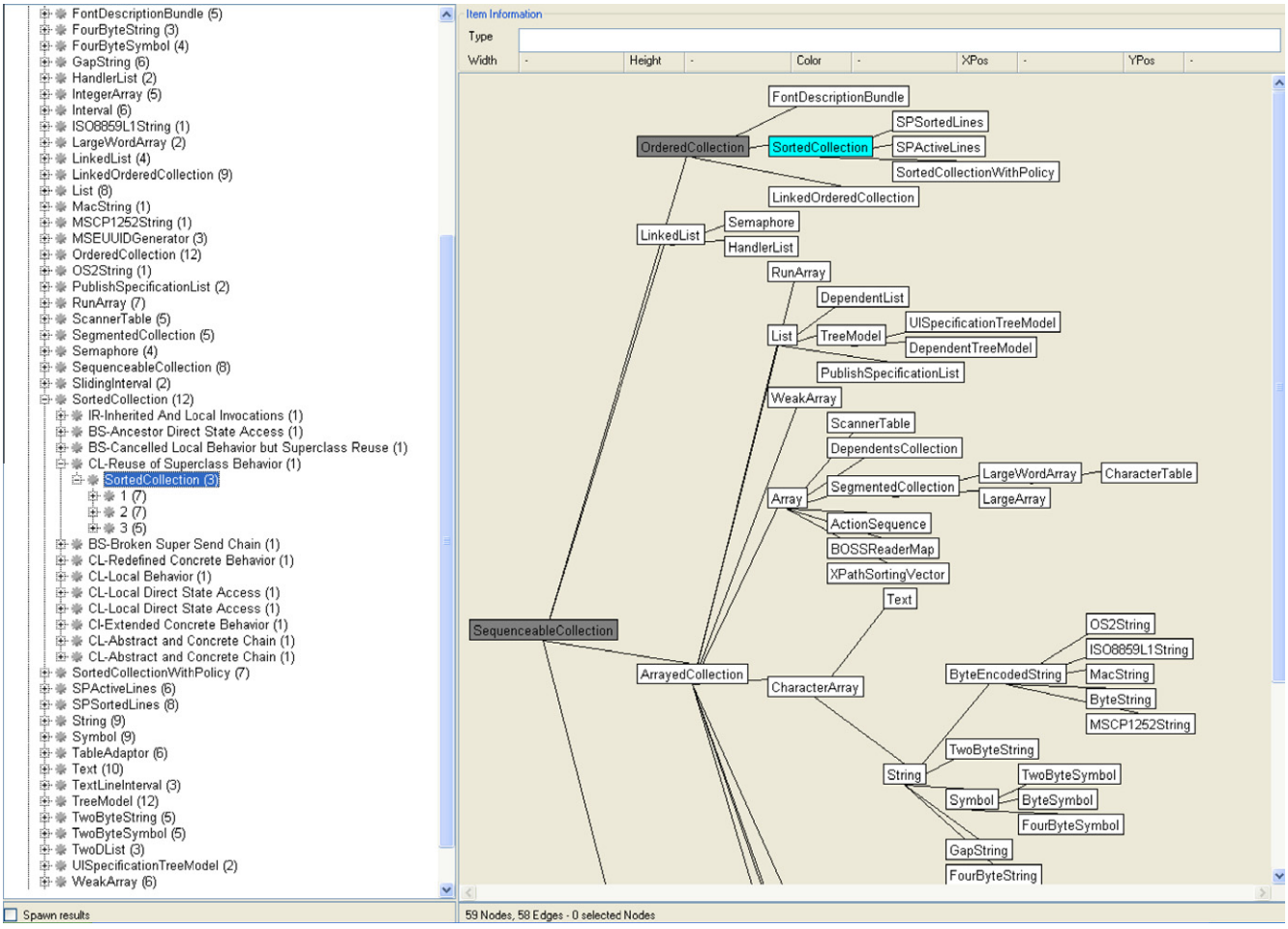

Fig. 9. Main window of the FoCARE methodology implemented in Smalltalk.

Table 3
Left: Commonly Identified *Classical* Schemas – Right: Commonly Identified *bad smell* and *Irregularities* Schemas.

Name Classical	Nr.	Name Bad smells	Nr.
Local Direct State Access	72	Ancestor Direct State Access	19
Local Behavior	69	Cancelled Local Behavior but Superclass Reuse	1
Template and hook	17	Abstracting Concrete Methods	8
Redefined Concrete Behavior	43	Cancelled local or inherited behavior	6
Extended Concrete Behavior	37	Broken *super* send chain	7
Reuse of Superclass/State Behavior	111	Irregularities	
Local Behavior overridden in Subclasses	29	Inherited and Local Invocations	15
Abstract and Concrete Chain	10	Unused Local Behavior but Superclass Reuse	3
		Accessor Redefinition	4

"Bad Smell": Ancestor Direct State Access. This schema identifies classes that access (read or modify the values of) the state of an ancestor class without using the accessors defined in the ancestor classes. We identified 19 classes that are part of the subhierarchies determined by **GeneralNameSpace**, **Dictionary**, **OrderedCollection** and **LinkedList**. In most of the cases, the classes are accessing state of the immediate superclass, but in the subhierarchy of **OrderedCollection** we detected several classes that access state of ancestors higher up in the chain of their superclasses. This is a not good coding practice since it introduces an unnecessary dependency on the internal representation of ancestor classes, and thereby violates encapsulation. Note that this happens in particular in our case study, because there are no protected visibility modifiers in Smalltalk. Fig. 10 illustrates this schema.

"Bad Smell": Cancelled local or inherited behavior. This schema identifies concrete local or inherited methods that are invoked via a *self* send in a class or its superclasses but are then cancelled in subclasses. Method cancellation is a sign that inheritance is being applied purely for code reuse purposes, without regard for subtyping. Since methods of the superclass calling the cancelled methods can still be called on the cancelling class, this may lead to runtime errors. In the **Collection** hierarchy it occurs in the

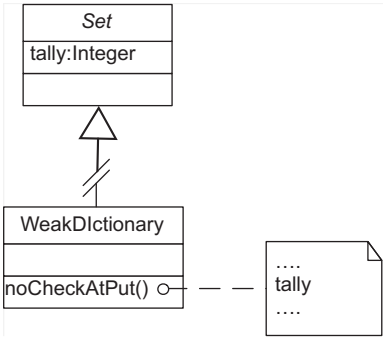

Fig. 10. Ancestor Direct State Access.

subhierarchies of **SequenceableCollection** and **OrderedCollection**. Fig. 11 illustrates this schema.

Irregular: Inherited and Local Invocations. This schema shows methods that are invoked by both *self* and *super* sends within the same class. In certain cases, this schema can be a *good practice* coding (as shown in Fig. 12), but a problem occurs when the *super* sends are invoked from a method with a different name. This special case of the schema occurs in the classes **LinkedOrderedCollection**, **LinkedWeakAssociationDictionary** and **XMainChangeSet**. All these classes have a special form: the class overrides a method *m*, and *m* invokes a method named *own-m* via *self* send, and this last method calls *m* via a *super* send implemented in the superclass. Fig. 13 illustrates this schema. This is an *irregular* case of the schema *Redefined Concrete Behavior* because the class is overriding the superclass behavior but is indirectly using the superclass behavior.

5.2. "Class-based" view on SortedCollection

With the *global view* we analyze a class hierarchy, but our approach also helps us to analyze how a class is built in the context of its superclasses and subclasses.

We chose to analyze the class **SortedCollection** (a subclass of **OrderedCollection**), because it is a good example where several schemas are overlapped in the same class, and this fact helps the developer to understand how it was built and how it works. A **SortedCollection** is an ordered collection of elements, using a sorting function for the elements order. The class has one attribute **sortBlock** which holds the sorting function; it has one class variable **DefaultSortBlock** that holds the default sorting function. As a subclass of **OrderedCollection**, it inherits two instance variables **firstIndex** and **lastIndex** and an indexed variable **objects**. Regarding its methods, it defines 10 methods and overrides 19 methods from the 403 inherited.

In this class we identify twelve different instances that correspond to four different schemas that involve this class.

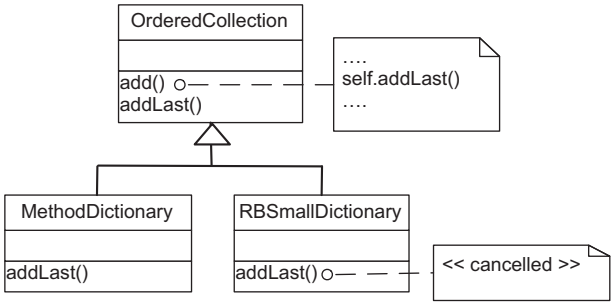

Fig. 11. Cancelled inherited behavior.

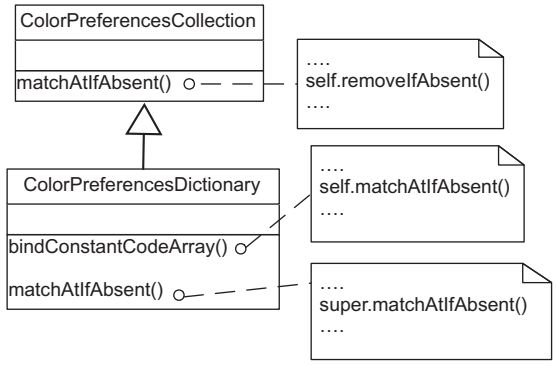

Fig. 12. Inherited and Local Invocations – Case 1.

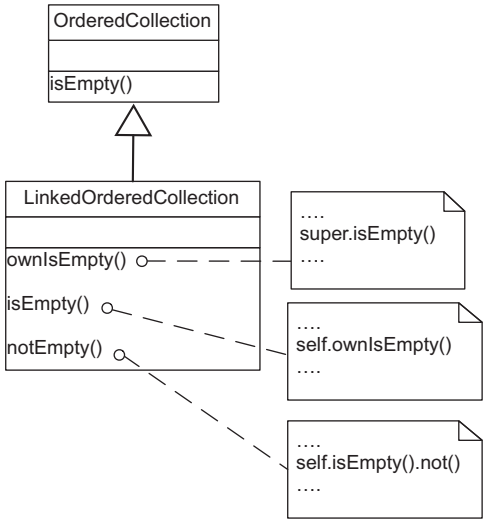

Fig. 13. Inherited and Local Invocations – Case 2.

Within the *Classical* category we report one case.

- *Reuse of Superclass/State Behavior:* This schema shows us that the class **SortedCollection** calls via *self* the methods **copyEmpty**, **insertBefore**, **reverseDo**, **asArray**, **isEmpty** and that these methods are not defined in the class itself but in different superclasses. Specifically, we see that the methods **copyEmpty**, **insertBefore** and **isEmpty** are defined in the class **OrderedCollection**, **reverseDo** and **asArray** are defined in the class **SequenceableCollection**. We see which are the superclasses that determine the behavior of the class. Fig. 14 illustrates this schema.

Within *"Bad Smell"* category, we report two cases:

- *Broken super send Chain:* This schema identifies methods that are extended (i.e., performing a *super* send) in a class but redefined in their subclasses without calling the overridden behavior. **SortedCollection** has an extension contract with its direct superclass calling the methods = and **representBinaryOn:** via a *super* send from the methods with the same

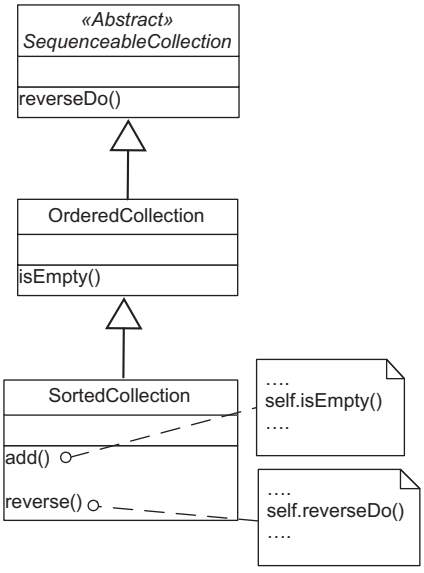

Fig. 14. Reuse of superclass behavior.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

name. But the methods in the subclass **SortedCollectionWithPolicy** do not invoke the corresponding inherited method defined in **SortedCollection** via a *super* send. This means that the subclass is overriding the method and is defining its own behavior but it is not respecting the behavior predefined by its superclass. Such a behavior can lead to unexpected results when the classes are extended without a deep knowledge of the implementation. Fig. 15 illustrates this schema.

- *Cancelled Local Behavior but Superclass Reuse:* This schema shows that the method *addAll* is called *via a super send* and this method is defined in the immediate superclass **OrderedCollection**, meaning that the class is reusing the behavior of the superclass. But this method is also implemented in the class **SortedCollection** but the behavior is *cancelled*. Although it is not a good practice, it seems a normal situation because the elements in a *sorted collection* cannot be added to the end of the collection, but only in a predefined position defined by the sorting function of the class. As we said previously, this is a case where the inheritance is used as code reuse without regarding *subtyping*. Specifically, this means that **SortedCollection** is a kind of **OrderedCollection** but not all the inherited methods can be applied. Fig. 16 illustrates this schema.

Within the *Irregular* category, we only found one case:

- *Inherited and Local Invocations:* This schema shows that the method **copyEmpty** is invoked with *self* sends and *super* sends in the class **SortedCollection**. It is implemented in the class itself, has an implementation in the superclass **Collection** and an implementation in the subclass **SortedCollectionWithPolicy**. When checking the code, we see that only **copyFromTo** calls **copyEmpty** within **SortedCollection**, and that the method **copyEmpty** has only one line of code, which makes only a super call to its overriden superclass method. From the viewpoint of internal behavior within **SortedCollection**, **copyEmpty** (with only one codeline) adds a level of indirection to the call flow considering that the behavior of this method is determined by the superclasses. A further analysis can help us to determine if **copyEmpty** in **SortedCollection** could be deleted and how we should adapt the behavior of **copyEmpty** in **SortedCollectionWithPolicy** and **copyFromTo** in **SortedCollection** to call directly the method **copyEmpty** defined in **Collection**. Fig. 17 illustrates this schema.

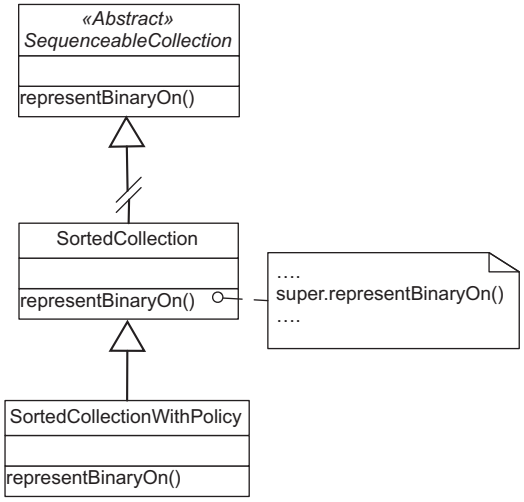

Fig. 15. Broken *super* send chain.

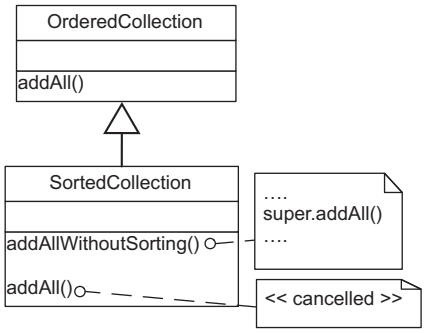

Fig. 16. Cancelled Local Behavior and Behavior Reuse of Superclasses.

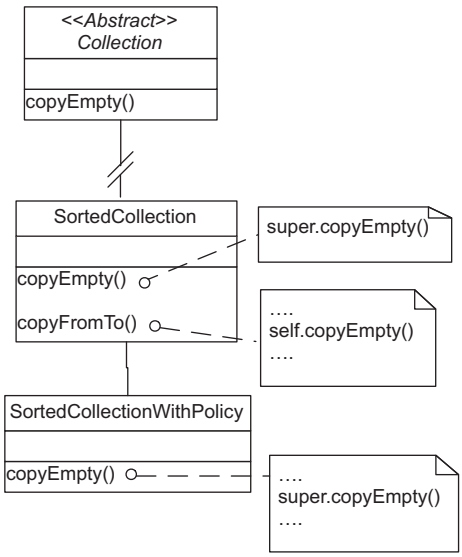

Fig. 17. Inherited and Local Invocations.

The identified schemas in our approach provide another view on the class. They present some unanticipated dependencies between the methods of the classes and their relationships in the hierarchy. Summarizing, our analysis confirms that although the **Collection** hierarchy offers many of the most powerful classes in *Smalltalk*, it is a rich and complex case study. However, it is not easy to modify or extend it, because our *schemas* show that the main used building mechanisms are subclassing and code sharing. Inheritance used for implementation reuse can lead to fragility in the design of the class hierarchies [56]. An analysis of relevant refactorings can be proposed to improve how the classes should be defined in this class hierarchy. The case study shows that our approach helps in identifying unanticipated contracts and relationships between classes within an inheritance hierarchy. It reveals patterns that are otherwise complex to spot due to method cancellations, local redefinitions and the yoyo effect.

6. Discussion and threats to validity

In this section, we discuss some issues related specifically to the use of FCA and we analyze some threats to validity of the current approach.

6.1. Discussion

Partial usage of lattice: We pointed out that once the concepts and the corresponding lattice are built, each concept represents a

group of invocations and accesses that relate a group of classes. But not all the concepts are relevant, and we keep only the meaningful concepts. There are mainly three points to take into account. First, if we analyze the position of those concepts in the lattice, we see that most of them are located in the lower part of the lattice, and we filter out the concepts located in the middle and upper part of the lattice. This is because the concepts in the lower part of the lattice contain more properties (inversely, few elements with those commonalities) than concepts higher up in the lattice (inversely, more elements with fewer commonalities). The lower concepts provide more "interesting" information (based on the combination of properties) and allow us to map them to non-trivial schemas of classes in a hierarchy. Secondly, we only use 64 of 174 concepts in total, meaning that just 1/3 of the lattice is used. Finally, we note that, in this particular application of FCA, we do not exploit the possible relationships (defined by the partial order) between the schemas (mapped from the concepts) because to our knowledge so far, there is no corresponding relationship between our Hierarchy Schemas in terms of software engineering concepts. However, we build the lattice because we need to calculate the reduced extent of the concepts to be sure that the invocations and accesses appear in only one concept, and participates in the most restrictive schema in the class hierarchy.

Compact number of elements and properties: In Section 3, we saw that we need to map the model entities (in our specific case, the invocations and accesses) to *FCA elements* and build different properties based on them. Due to a performance limitation imposed by the FCA algorithm [45], we reduce dramatically the number of FCA elements keeping only one representative of an invoked method and accessed attribute in a class. We compute the concepts and the lattice without losing information about the class hierarchies, and we reduce also the computing time from around 1 hour to 10 minutes compared to the approach presented in previous work [4].

FCA as a classification mining tool: As expressed previously, we focus on analyzing (groups of) related dependencies between classes in a hierarchy based on *self* and *super* invocations. Without FCA, we should generate all the possible combinations of (explicit and implicit) dependencies between classes, resulting an exponential number of candidates contracts. Each combination can define a known or an unexpected contract that should be identified and verified if it occurs in the class hierarchy. For example, if we had three properties P_1, P_2 and P_3, without FCA we should generate the following sets $\{P_1\}$, $\{P_2\}$, $\{P_3\}$, $\{P_1, P_2\}$, $\{P_1, P_3\}$, $\{P_1, P_4\}$, $\{P_2, P_3\}$, $\{P_2, P_4\}$, $\{P_3, P_4\}$ and $\{P_1, P_2, P_3, P_4\}$. Once the sets are generated, we should test if every invocation to a method and access to an attribute fulfils or not any set of properties. We should also check if we can group elements with same set of properties, and also if there is any relationship between the different identified sets. Clearly, we see that the process can be expensive in terms of performance and processing, and FCA seems a better choice in our context to avoid these difficulties.

FCA vs. query-engine: One of the main results of this approach is a *catalog* of schemas to characterize a class hierarchy. As we see in Section 4, each schema is the interpretation of a *conjunction* of properties in the concepts. Each schema can be expressed as a logic predicate (mapped from the properties) and a query engine can be run in a class hierarchy to identify the occurrences of the different schemas. The main difference between the use of FCA and a query-engine is that FCA helps us mainly to discover *previously unknown* schemas introduced in a class hierarchy, because we do not know in advance which are the properties that define them. In the case of the use of a query approach, we need to know which are the different properties that characterize a schema. We consider that the two approaches are complementary, because the catalog of schemas can be complete after the analysis of several class hierarchies, and in that case, the application of query-engine is more suitable than FCA. We can have an *adapted architecture* where the step *FCA Mapping* is converted into *Query-based Mapping*, and the steps *ConAn Engine* and *Post-Filtering* are not present any more.

6.2. Threats to internal validity

Use of language-dependent properties: In spite of the fact that our approach is language independent, how the properties are extracted from source code depends on language-specific mechanisms or idioms. Let us enumerate some of them in the analysis of class hierarchies in Smalltalk code (presented in this paper) compared to some initial experiments in Java [14].

- The property *m is cancelled* in *C* is used to indicate that a subclass cancels a method defined in a superclass. It can be detected when the method body contains only the call `self shouldNotImplement`, which generates a runtime error in execution time. The property *m is abstract* in *C* is detected when the method body contains only the call `self subclassResponsibility`. When applying the approach to another object-oriented language, we have to adapt the extraction of properties, and should consider specific properties regarding the chosen language. In Java, we have seen that the properties *m is cancelled* in *C*, *m is abstract* in *C*, *C accesses a* and *invoked via self* have the same meaning as in Smalltalk code analysis but are extracted in a different way [14]. We chose to define the property *m is cancelled* in *C* when the method body only raises an exception. The property *m is abstract* in *C* is detected using the *abstract* modifier of the method declaration. The property *C accesses a* can be set up with and without accessors as in Smalltalk code, except that in Java we can have the direct access to the state with the expression `this.<variable name>`. Finally, the property *invoked via self* can be extracted by looking for the call where the receiver is the keyword `this`, but also for calls where there is no explicit receiver. In this case, it is implicit that the receiver is `this`.
- There are several properties specific to Java that are worth mentioning. Firstly, we have to build properties to represent the restrictive visibility of the methods (public, private, protected). We also have to map *interfaces* and *anonymous classes* to our FCA approach.
- The added properties increase the catalog of possible Hierarchy Schemas detected in Java code, and helped us to identify which Hierarchy Schemas are common to several object-oriented languages, and which ones are specific to a particular object-oriented language. For example, in Smalltalk all methods are public (there being no visibility modifiers). The schema called Cancelled Local Behavior but Superclass Reuse is defined with the set of properties *invoked via super, is cancelled locally, is concrete in ancestor* C_n *(n: 1 ..m)*. If we consider Java code, the same schema is represented with the set of properties *is public, invoked via super, is cancelled locally, is concrete in ancestor* C_n *(n: 1.. m)*. As we see, the only different property between both definitions is *is public*, and this property is needed in Java code due to the presence of visibility modifiers, and we need them to identify this characteristic in the methods. However, this property is implicit in the detected schema in Smalltalk code. In spite of this slight difference, we can infer that Cancelled Local Behavior but Superclass Reuse is a common schema in both Java and Smalltalk code. However, any schema that contains any property such as *is protected* or *is private* clearly is specific to Java or C++ code. Busch et al. present some preliminary results [14].

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

Table 4 shows some initial results from analyzing the Java Collection Package with 162 classes. From our initial analysis, the use of interfaces and anonymous classes hides the occurrences of other schemas in this case study. We have to deepen our analysis including Java building mechanisms.

Mapping from concepts to schemas: Of the 64 concepts we identified as "interesting", we derived 15 dependency schemas. This means that in most cases, a schema is represented by several concepts, meaning that a schema can be described by different combinations of properties. The mapping policy is decided by the reengineer, meaning that when we interpret the contents of the concepts, we must decide which are concepts corresponding to the different schemas. For example, the schema *Reuse of Superclass/State Behavior* is represented by three types of concepts because there are three different combinations of properties to describe the mentioned schema (as shown in the table of Hierarchy Schemas). On the other hand, the schema *Local Behavior* is represented by just one concept. In other cases, one schema could represent a *good* or an *irregular* design practice. In this specific case, we see that the schema *Inherited and Local Invocations* is *irregular* only when the *super* sends are invoked from a method with a different name.

Building elements and properties. We map the attributes accesses and methods calls directly from the metamodel FAMIX [60]. The choice of properties requires some analysis (done in the iterative phase *Modelling* of FoCARE), because we need to cover the different possible inheritance dependencies of the elements. The properties *invoked via self* and *invoked via super* are mapped directly from the metamodel. The rest of the properties are calculated based on the dependencies expressed in the metamodel.

Number of concepts: The policy of mapping from concepts to schemas is manual in initial experiments, until their identification can be automatized. To determine the correct mapping, we have to go through some cycles of *Interpretation* using only parts of the case study. The main drawback in this phase is the number of generated concepts in the lattice, especially when there are not so many elements that share common properties and this can generate an explosion in the number of concepts. This happens because in the first experimental *Interpretation*, we have to analyze the concepts manually until we get the right definition of each schema, and this is not trivial when the number of concepts is large.

6.3. Threats to external validity

We note that the approach can succeed or fail depending on several factors.

"Non-invoked" Methods: Our approach is limited to analyzing methods and attributes that are effectively used in the context of the class hierarchy. If there are methods that are defined in some class but are not invoked in the class itself or in any subclasses or in any superclasses, those methods are not included in our analysis. Clearly, we lose some information about the classes in the hierarchy, because we only concentrate on usage of behavior and state of the class.

Table 4
Results in Java Collection Package. Left: Commonly Identified *Classical* Schemas – Right: Commonly Identified *bad smell* Schemas.

Name Classical	Nr.	Name Bad smells	Nr.
Redefined Concrete Behavior	3	Ancestor Direct State Access	2
Extended Concrete Behavior	1	Cancelled Local Behavior but Superclass Reuse	3
Reuse of Superclass/State Behavior	45		

Incrementing the catalog of schemas: Although we consider our approach to be mature enough to detect interesting hierarchy schemas, during new applications of the approach to other Smalltalk class hierarchies or to other object-oriented languages, we may discover new schemas to augment the hierarchy schemas catalog. Note that this threat is not negative since we identify them and simply build incrementally our catalog. This illustrates the usefulness of FCA as a classification mining tool. When we reach a fix point for given schema catalog (meaning that when running the approach there are no new identified Hierarchy Schemas), FCA ceases to be useful as a mining tool and can be replaced by a query-based engine. This replacement implies only to change a layer in the our architecture. Thus, we show that the architecture is adaptable to changes like this one.

Choice of properties and elements: The choice of properties and elements is critical in this approach. Without adequate elements and properties, the approach cannot produce meaningful concepts, and therefore we cannot detect any practical schema. Therefore, it is crucial to test them in parts of the case studies going through several cycles of *Modelling* and *Interpretation* of the phases of the FoCARE architecture. The reengineer can decide if the conceptual mapping the useful for analyzing a piece of software (in our specific case, class hierarchies).

Good programming style: When applying this approach, we focus on detecting not only good practices in coding, but also *bad smells* and *irregularities* in class hierarchies, and we propose *refactorings* to improve the code. If the class hierarchy is well designed, this approach only shows good practices in coding. We consider that applying this approach is too expensive to use merely to confirm that the class hierarchy is well-designed.

Small hierarchies: Similarly, in the case of small hierarchies, applying this approach is too expensive compared to analyzing it manually. In this paper, we have shown how useful is FoCARE in **Collection** class hierarchy, which has around 100 classes.

Exponential number of concepts and schemas: Another critical issue from the FCA viewpoint is the large (eventually, exponential) number of concepts that FCA can generate. This situation can arise if the properties are too specific for a small set of elements and/or the incidence table density is low, meaning that there are only few pairs of (element, property) that are valid. From a coding perspective, this means that the dependencies are dispersed and do not recur in the class hierarchy. With too many results, it is hard to analyze new schemas because they are interpreted manually (before including them in the catalog), and the results depend on the knowledge of the reengineer.

Performance of FCA algorithm: As mentioned previously, the algorithm is also an essential factor in this approach. The performance can be affected by two circumstances. The first one is related to implementation issues. It is known the performance limitation analyzed by Kuznetsov and Obëdkov [45], so the reengineer should be careful when implementing it, or should use an existing implementation that exhibits good performance. The second circumstance is when the incidence table has low density, and this can generate an exponential number of concepts, and the calculation of partial order can be time-consuming.

Biased evaluation: As we have developed the approach and carried out the experiment ourselves, the results may be biased. This effect is mitigated by the fact that we have chosen to analyze the *Smalltalk* Collection hierarchy as a case study which has previously been studied by several other researchers [11,17,35,46].

7. Related work

Within the related approaches, we identify three main fields: *Understanding (and evolution of) class hierarchies*, *Query-based en-*

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

gines to understand object-oriented applications and *Detection of design patterns*.

7.1. Understanding (and evolution) of class hierarchies

Various existing approaches have explored techniques to support the understanding (and evolution of) class hierarchies. We survey them according whether or not they use Formal Concept Analysis (or related structures) in their approaches.

Cook [17] proposes to reorganize the Smalltalk-80 collection class library based on the hierarchy of interfaces, which is independent of inheritance. With this approach, he can identify several problems such as inherited methods that violate the subclass invariant; methods that have the same name but unrelated behaviors; methods that have the same (or related) behavior but different names. Once the problems are identified, he can suggest improvements to the analyzed class hierarchy.

Black et al. [11] report on their experience refactoring the Smalltalk Collection classes using traits [64]. After manually identifying duplication of code and lack of uniformity in the protocols of the classes, they could refactor the hierarchy with traits and reduce the number of methods by approximately 10%. The new hierarchy improves maintainability and reuse of the code base.

Based on the lack of adequate documentation and the absence of mechanisms to manage the propagation of changes of evolving class hierarchies, Steyaert et al. [72] introduce the concept of *reuse contracts*. The goal is to separate a particular kind of design information from the implementation. This is achieved by recording the protocol (*specialization interface*) between classes and its subclasses. When classes change, this (explicit) documentation allows one to identify which *contracts* are no longer valid and what part of (analyzed) class hierarchy should no longer be trusted. Focusing also the class evolution, Casais [15] analyzes class hierarchies and extracts *design flaws* when a (new) class may refine or override the properties inherited from its ancestors. With an incorrect use of inheritance mechanisms, some defective structures can appear in the class hierarchy. To avoid them, the redefinitions are analyzed and are adapted (if necessary) to improve subclassing patterns using *decomposition* or *refactoring*. The approach has been validated in Eiffel systems.

The following four approaches use visualization techniques. *Program Explorer* [47] proposes to query and visualize both dynamic and static information of classes via simple graphs to understand and verify hypotheses about function invocations, object instantiation and attribute accesses. This approach is validated in C++ applications. Using basic graph visualizations to represent various relationships and navigation features, Mendelzon and Sametinger [53] show that they can express metrics, constraints verification, and design schema identification in large-scale software systems. Ducasse and Lanza [48] introduce the notion of a *Class Blueprint*, a semantically enriched visualization of the internal structure of classes, which allows a software engineer to develop a mental model of the classes, and offers support for reconstructing the logical flow of method calls. Identifying *visual patterns*, the engineer can identify not only good hierarchy practices, such as classes that add, extend, or override inherited behavior of superclasses, but also *bad smells*, such as duplicated code between classes. With this approach, the engineer can perform the analysis and browse the code to validate his hypotheses. Focusing on class hierarchy understanding, Denier and Sahraoui [21] propose a similar approach. They provide a compact view of class hierarchies using a custom Sunburst layout. Then, they map class properties to graphical attributes of a 3D visualization, and using metrics to characterize similar children classes, they derive a set of *visual patterns*. These patterns can identify good and bad practices introduced in the analyzed class hierarchy.

Considering the useful potential specialization/generalization of the lattices, several researchers have also applied FCA to the problem of understanding (and reengineering) class hierarchies.

In reengineering class hierarchies, Godin et al. [33] categorize the existing work according to the structure of the output hierarchy of the approaches. Snelting et al. [70,69] use *a (concept) lattice* as a final result of their analysis. Godin et al. [34,35], Dicky et al. [22,23], Huchard et al. [41] and Moore [57] use *Galois subhierarchy*[4] as a final results of their hierarchy. To this classification, we added the approach of Falleri et al. [28], based on a derived structure named RCA.

Snelting et al. [70,69] presented a methodology to find design problems in a class hierarchy by analyzing the usage of the hierarchy by a set of C++ applications. They analyzed a class hierarchy making the relationship between class members and variables explicit. They were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class or where a splitting of a class is needed. As a result, they propose a new class hierarchy that is behaviorally equivalent to the original one. The implementation of this approach for refactoring Java class hierarchies is proposed by Streckenback and Snelting [73] using KABA. Additionally, the transformed hierarchy can then be subject to further manual refactorings, while the semantics are guaranteed to be preserved. The new code contains the same statements as the original code, except that the hierarchy has changed and for all variables a new type (*i.e.*, class) has been computed.

Godin et al. [34,35] proposes an approach to build an initial class hierarchy from a set of class specifications, or reorganize an existing one or detect inconsistencies following class updates using concept lattices and related structures. They show how Cook's [17] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. Their approach has added a set of metrics to measure redundancy, specialization and aggregation complexity and deviation from specialization. Therefore, they could compare variants of concept lattices among themselves, and to other manually or automatically-built class hierarchies. They have applied their approach to the Smalltalk Collection hierarchy, and also to a set of class specifications pertaining to telecommunication network management functionalities.

The GURU approach developed by Moore [57] analyzes classes (eventually without inheritance links) in Self and infers a hierarchy with no duplication of features (instance variables and methods). The new hierarchy includes replacement classes defining or inheriting exactly the same sets of features as defined by the original classes. Thus, the hierarchies are produced and based only on maximizing sharing and minimizing duplication of features (mostly methods) of classes.

Preserving a "maximal factorizing" of class properties, Huchard et al. [41], Dicky et al. [23] and Leblanc [50] focus on the automatic insertion of classes into inheritance hierarchies. They use two incremental algorithms (ARES in first two cases and CERES in last case) to factor out features using overloading and overriding. They guarantee an optimal refactoring of features of classes by using a Galois subhierarchy, which also provides a partial order on attributes and methods (signatures). To measure the quality of the generalization/specialization in the *new* class hierarchy, Roume [63]

[4] By analogy with normalization for database design, the concept lattice can be considered as a kind of normal form for the design of class hierarchies. Some of the generalizations of the concept lattice are empty in that they do not possess their own attributes or objects: all their attributes appear in at least one super-concept (inheritance) and dually, all their objects appear in a sub-concept (extension inclusion). These concepts could be eliminated without loss of information thus leading to a structure called a Galois subhierarchy. This structure is not necessarily a lattice but, when interpreting its nodes as classes, it is optimally factored, consistent with specialization, while defining a minimal number of classes.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

and Dao et al. [18] introduce a set of metrics that measures refactoring at the level of classes and features. With this set of metrics, they can quantify the improvements carried out by a reconstruction tool and highlight design defects connected with feature refactoring, such as features that are obviously redundant or classes that are useless.

Using Relational Context Analysis (RCA), a derived structure from FCA, Falleri et al. [28] propose a generic approach for normalizing class hierarchies that follows the work of Huchard et al. [41], Dicky et al. [23] and Leblanc [50]. This approach can be applied to any class model that can be described by a metamodel. Some previous work [6] showed that FCA was able to deal with only specialization/generalization of classes. Falleri et al. [28] use RCA in this approach in order to deal with entities described by binary attributes and by relations with other entities. RCA will allow to support constructions, such as invariant or covariant method redefinition and covariant attribute redefinition. This approach is validated in applications based on Ecore and Java.

Within the context of traits [64], Lienhard et al. [51] propose a semiautomatic approach to identify traits in an existing class hierarchy using FCA. Their tool proposes a refactoring of the class hierarchy with traits that preserves the original behavior of each of the classes. Their approach is carried out in two main steps. First, they analyze the protocol of classes to identify the potential traits, and then they analyze the invocation relationships between methods to detect fine-grained traits. The case study shows that they obtain similar results to those obtained manually by Black et al. [11].

7.2. Query-based engines to understand object-oriented applications

Using the notion of *Micro Patterns* in Java code, Gil et al. [31] extract the various kinds of features that a Java class may have, and how they can be related. Then, the relationships are translated into a condition on the code, and classes are inspected to match those conditions. Manual inspection of the code of these classes leads to refinement, removal, merging or splitting of the definitions until an acceptable catalog of micropatterns is built.

Using queries that define *Inheritance* or *Composition Template Methods*, Schauer et al. [65] focus on identifying hook and template methods, which capture the flexible parts of an application domain. With tool support named SPOOL, they provide a visualization of the hotspots, so it helps the programmers to identify key design abstractions as well as their implementation style, based on inheritance or composition.

Analyzing class hierarchies with criteria of code or interface reuse, Mihancea [54,55] introduce the concept of *comprehension pitfall* as a design situation in which the polymorphic manipulation of a design entity (*e.g.*, method) can be easily misunderstood. The pitfalls are defined using metric-based rules to support their automatic detection. They applied their approach to three medium-sized Java programs, using an iterative approach where the manual analysis has helped to maximize the precision of detection rules, to be able to detect meaningful pitfalls.

Focused on testing of object-oriented applications, Ducasse et al. [25] propose the logic representation of program execution and the specification of tests as logic queries using Smalltalk Open Unification Language (SOUL), a logic engine implemented in Smalltalk. Thus, complex sequences of message exchanges, sequence matching, or expression of negative information are expressed in compact form. Although the implementation is in Smalltalk, the approach itself is not specific for Smalltalk but can be applied to other languages as well, such as Java.

Analyzing software evolution, Lanza et al. [49] propose a flexible query-engine to perform queries on different versions of a system. The queries make the inheritance operations explicit, such as introducing a class on top of a large hierarchy, classes that have been merged, renamed or pushed up one hierarchy level, or any entity which has been added to or removed from software at a certain point.

Ciupke [16] presents a technique for analyzing legacy code, specifying frequent design problems as queries and locating the occurrences of these problems in a model derived from source code. He can check violations of a number of well-known design rules in existing source code taken from several case studies, showing that the task of problem detection in reengineering can be automated to a large degree, and that the technique presented can be efficiently applied to real-world code.

In Krämer and Prechtel's approach[44], the patterns are stored as Prolog rules. Their Pat tool takes the meta-information directly from the C++ header files and queries them. Similarly, Wuyts [78] reasons about and extracts a system's structure using SOUL, and by developing a declarative framework aimed at reasoning about Smalltalk code.

Focusing on understanding how a framework works, Lange et al. use an interactive visualization of design patterns to understand the underlying the software architecture [47]. They represent both static and dynamic information as logic facts to generate interactive design views.

7.3. Detection of design patterns

Design patterns are used as ways of communicating design information. Beck and Johnson [9] show that patterns can be used to derive an architecture, and the resulting description makes it easier to understand the purpose of the various architectural features.

Gueheneuc, Mens and Wuyts propose a framework to compare design recovery tools [38], to understand their differences, to ease replication studies, and to discover what tools are lacking.

Shull et al. propose a method to manually identify workable domain-specific design patterns and create customized catalogs of the identified patterns [68]. Brown [12] presents a tool to detect design patterns in Smalltalk environments. He explains how to deal with the typeless language Smalltalk. Keller et al. [42] present an environment for the reverse engineering of design components based on the structural descriptions of design patterns. Their validation is made with SPOOL on three large-scale C++ software systems. Bergenti and Poggi provide critiques about the design patterns identified in UML documents [10]. Philippow et al. promote a design pattern-based approach to reconstruct the reference architecture of a product line [62].

Seemann et al. [66] use a compiler to generate graphs from the source code. This graph acts as the initial graph of a graph grammar that describes the design recovery process. The validation is made with respect to well-known design patterns such as *Composite* and *Strategy* in the Java AWT package.

Niere et al. [58] provide a method and a corresponding tool which assist in design recovery and program understanding by recognizing instances of design patterns semi-automatically. The algorithm works incrementally and needs the domain and context knowledge given by a reverse engineer. An evaluation of the approach is made with the Java AWT and JGL libraries.

Albin-Amiot et al. [1] show how to automate the instantiation and detection of design patterns. To cope with these objectives, they define a *Pattern Description Language* to describe design patterns as first-class entities. Thus they can manipulate and adapt design patterns models to generate source code.

One of the main problems in pattern identification is the size of the search space. To reduce it, Wendehals [76] and Heuzeroth et al. [40] combine static and dynamic analysis, the first one reducing the search space of the second one: the static analysis searches for sets of candidates that respect the static structure of the design

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

pattern, while the dynamic analysis monitors candidates and checks whether the observed interactions satisfy the behavioral rules of the design patterns. Gueheneuc et al. used explanation-based constraint programming to report problems when failing to identify design patterns [37]. Antoniol et al. propose a multi stage reduction strategy: software metrics and structural properties computed on design patterns become constraints that design pattern candidates must satisfy [3]. Guenehec [39] reduces the search space using metrics to define design pattern fingerprints of the design pattern participants. A design pattern has several design variants and can be implemented in different ways; Niere [59] overcomes both problems with fuzzy logic, and Wendehals [76] rates instance candidates with fuzzy values to support inexact mismatch.

Differences with existing approaches. Our approach exhibits two main differences compared to existing approaches: (1) input information and (2) discovery of *new* patterns (in our case called *schemas*).

Regarding the input information, most of the approaches (related to understanding class hierarchies) take into account which selectors are implemented by which classes (*interface*). They do not consider behavioral information (i.e., based on *self* and *super* sends) or usage of the state defined in the classes. As shown in this paper, such information helps us to identify different *behavioral* and *state dependency schemas*. With these schemas, we evaluate the reuse of methods and state defined in classes, and we discover different *design* decisions used in building the class hierarchies. Black et al. [11], Steyaert [72] and Lienhard *et al.*, [51] have similarities to our approach because they use behavioral information of the class hierarchies. In the case of Black et al. [11] and Steayaert [72], the information is extracted manually and does not depend on the language. In Lienhard et al. [51], the behavioral information is focused specifically on classes where potential traits can be extracted; it is not a global analysis as in our approach.

Regarding the discovery of *new* patterns, the main difference with the existing approaches related to query-based methodologies and detection of design patterns is that in all the approaches the user must know in advance which is the definition of the relationships or the software artifacts one is looking for in the target system. For example, to detect design patterns, one needs to know how the design pattern is defined structurally to express it as a query (*e.g.*, as rules in Prolog). In our approach, we work with simple relationships and properties of the source code, and we infer the definition of the *schemas* or *patterns* using FCA.

8. Conclusion and future work

In this paper, we show how the automatic generation of schemas using FCA helps us to discover different implicit and undocumented dependencies in class hierarchies in terms of the behavior and state usage. The categorization of these schemas into *good*, *irregular* and *bad* design decisions helps us to:

- Generate the first *conceptual model* of a hierarchy.
- Localize where different irregularities or problems occur in the implementation of a class hierarchy.
- Identify which are the main constraints that a specific class has in the context of a hierarchy. When a developer wants to use or extend a class, he needs to understand which are the different dependencies a class has regarding its subclasses and superclasses. These dependencies include if the class is overriding or reusing the behavior inherited from the superclasses or if the class has template methods and hooks that its subclasses should implement.

As a summary, we conclude that specific object-oriented reengineering tasks (such as understanding how a system is built) can be solved using techniques coming from graph theory (such as Formal Concept Analysis), and this technique gives the possibility of discovering (un)expected contracts between the classes. This FCA-based approach can be complemented with existing approaches (such as metrics) to enrich our analysis and our results.

As future work we plan the following next steps:

- Application of the approach to other class hierarchies to check if the catalog of schemas identified so far covers all the interesting possible cases or if we discover new cases of implicit contracts.
- Extended analysis comprising methods that are invoked and those that are not invoked but are declared in the classes. This kind of approach can measure how much information defined in the class hierarchy is used or not.
- Refinement of properties to obtain a bijective mapping from concepts to schemas, and thus, reduce the complexity of the lattice in terms of number of concepts.
- Analysis of relationships given by the *partial order* between the different schemas – mapped from the concepts in the lattice – to find a possible mapping in terms of software reengineering.
- Deep analysis of some initial experiments to other object-oriented languages, such as Java [14], where we have other building mechanisms, such as *interfaces*, *inner classes* and *anonymous classes*, and see how the approach can be adapted or modified to consider new properties. Surely, adding new properties will generate new schemas in our catalog.

Acknowledgements

We thank Simon Denier, Andres Fortier, Alejandra Garrido, Jannik Laval and Lukas Renggli for useful comments that helped us to improve this paper. The last author gratefully acknowledges the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, October 2008–September 2010).

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien. Instantiating and detecting design patterns: putting bits and pieces together, in D. Richardson, M. Feather, M. Goedicke (Eds), Proceedings of ASE '01 (16th Conference on Automated Software Engineering), IEEE Computer Society Press, November 2001, pp. 166–173.
[2] S.R. Alpert, K. Brown, B. Woolf, The Design Patterns Smalltalk Companion, Addison-Wesley, 1998.
[3] G. Antoniol, R. Fiutem, L. Cristoforetti, Design pattern recovery in object-oriented software, in 6th International Workshop on Program Comprehension (Ischia, Italy), 1998, pp. 153–160.
[4] G. Arévalo, Understanding behavioral dependencies in class hierarchies using concept analysis, in: Proceedings of Langages et Modeles à Objets (LMO'03). Hermes, Paris, January 2003, pp. 47–59.
[5] G. Arévalo, High Level Views in Object-Oriented Systems using Formal Concept Analysis. PhD thesis, University of Bern, Bern, January 2005.
[6] G. Arévalo, J.-R. Falleri, M. Huchard, C. Nebut, Building abstractions in class models: formal concept analysis in a model-driven approach, in: O.N.J.W.D.H.G. Reggio (Ed.), MoDELS 2006, LNCS (Lecture Notes in Computer Science), vol. 4199, Springer-Verlag, 2006, pp. 513–527.
[7] K. Auer, Reusability through self-encapsulation, in: Pattern Languages of Program Design, ACM Press/Addison-Wesley Publishing Co., 1995, pp. 505–516.
[8] K. Beck, Smalltalk Best Practice Patterns, Prentice-Hall, 1997.
[9] K. Beck, R. Johnson, Patterns generate architectures, in: M. Tokoro, R. Pareschi, (Eds.), Proceedings ECOOP '94 LNCS, vol. 821, Bologna, Italy, July 1994, Springer-Verlag, pp. 139–149.
[10] F. Bergenti, A. Poggi, Improving UML designs using automatic design pattern detection, in: 12th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2000, pp. 336–343.

[11] A.P. Black, N. Schärli, S. Ducasse, Applying traits to the Smalltalk collection hierarchy, in: Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03), vol. 38, October 2003, pp. 47–64.
[12] K. Brown, Design reverse-engineering and automated design pattern detection in Smalltalk. Master's thesis, North Carolina State University, 1996.
[13] T.A. Budd, An Introduction to Object-oriented Programming, Addison-Wesley, 1991.
[14] P.F. Busch, M. Pasqualino, Coding patterns in class hierarchies in object-oriented applications (in spanish). Master's thesis, Universidad Austral, Buenos Aires, Argentina, 2008.
[15] E. Casais, Managing class evolution in object-oriented systems, in: O. Nierstrasz, D. Tsichritzis (Eds.), Object-Oriented Software Composition, Prentice-Hall, 1995, pp. 201–244.
[16] O. Ciupke, Automatic detection of design problems in object-oriented reengineering, in: Proceedings of TOOLS 30 (USA), 1999, pp. 18–32.
[17] W.R. Cook, Interfaces and specifications for the Smalltalk-80 Collection Classes, in: Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications), vol. 27, ACM Press, October 1992, pp. 1–15.
[18] M. Dao, M. Huchard, T. Libourel, C. Roume, H. Leblanc, A new approach to factorization: introducing metrics, in: Proceedings of METRICS '02 (8th IEEE International Symposium on Software Metrics, IEEE Computer Society, 2002, pp. 227–236.
[19] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-oriented Reengineering Patterns, Morgan Kaufmann, 2002.
[20] S. Demeyer, S. Tichelaar, S. Ducasse, FAMIX 2.1 – The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.
[21] S. Denier, H.A. Sahraoui, Understanding the use of inheritance with visual patterns, in: Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, USA, October 2009, pp. 79–88.
[22] H. Dicky, C. Dony, M. Huchard, T. Libourel, ARES, Adding a class and REStructuring Inhertitance Hierarchy, in: BDA: 11èmes Journées Bases de Données Avancées, 1995, pp. 25–42.
[23] H. Dicky, C. Dony, M. Huchard, T. Libourel, On automatic class insertion with overloading, in: Proceedings of OOPSLA '96 (11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications), ACM Press, 1996, pp. 251–267.
[24] S. Ducasse, T. Gîrba, A. Kuhn, L. Renggli, Meta-environment and executable meta-language using Smalltalk: an experience report, Journal of Software and Systems Modeling (SOSYM) 8 (1) (2009) 5–19.
[25] S. Ducasse, T. Gîrba, R. Wuyts, Object-oriented legacy system trace-based logic testing, in: Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), IEEE Computer Society Press, 2006, pp. 35–44.
[26] S. Ducasse, M. Lanza, The class blueprint: visually supporting the understanding of classes, Transactions on Software Engineering (TSE) 31 (1) (2005) 75–90.
[27] A. Dunsmore, M. Roper, M. Wood, Object-oriented inspection in the face of delocalisation, in: Proceedings of ICSE '00 (22nd International Conference on Software Engineering), ACM Press, 2000, pp. 467–476.
[28] J.-R. Falleri, M. Huchard, C. Nebut. A generic approach for class model normalization, in: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), Italy, IEEE, September 2008, pp. 431–434.
[29] M. Fowler, UML Distilled, Addison-Wesley, 2003.
[30] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, Springer-Verlag, 1999.
[31] J. Gil, I. Maman, Micro patterns in java code, SIGPLAN Notices 40 (10) (2005) 97–116.
[32] T. Gîrba, S. Ducasse, Modeling history to analyze software evolution, Journal of Software Maintenance: Research and Practice (JSME) 18 (2006) 207–236.
[33] R. Godin, M. Huchard, C. Roume, P. Valtchev. Inheritance and automation: where are we now? in: A. Black, E. Ernst, P. Grogono, M. Sakkinen (Eds), ECOOP 2002: Proceedings of the Inheritance Workshop. University of Jyväskylä, June 2002, pp. 58–64.
[34] R. Godin, H. Mili, Building and maintaining analysis-level class hierarchies using galois lattices, in: Proceedings OOPSLA '93 (8th Conference on Object-Oriented Programming Systems, Languages, and Applications), vol. 28, October 1993, pp. 394–410.
[35] R. Godin, H. Mili, G.W. Mineau, R. Missaoui, A. Arfi, T.-T. Chau, Design of class hierarchies based on concept (galois) lattices, Theory and Application of Object Systems 4 (2) (1998) 117–134.
[36] O. Greevy, S. Ducasse, T. Gîrba, Analyzing software evolution through feature views, Journal of Software Maintenance and Evolution: Research and Practice (JSME) 18 (6) (2006) 425–456.
[37] Y.-G. Guéhéneuc, H. Albin-Amiot, Using design patterns and constraints to automate the detection and correction of inter-class design defects, in: Q. Li, R. Riehle, G. Pour, B. Meyer (Eds.), Proceedings of the 39th Conference on the Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, July 2001, pp. 296–305.
[38] Y.-G. Guéhéneuc, K. Mens, R. Wuyts, A comparative framework for design recovery tools, in: Conference on Software Maintenance and Reengineering (CSMR 2006), Los Alamitos CA, IEEE Computer Society Press, 2006.
[39] Y.-G. Guéhéneuc, H. Sahraoui, F. Zaidi, Fingerprinting design patterns, in: Working Conference on Reverse Engineering (WCRE'04), Los Alamitos CA, IEEE Computer Society Press, 2004, pp. 172–181.
[40] D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic design pattern detection, in: International Workshop on Program Comprehension, 2003, pp. 94–104.
[41] M. Huchard, H. Dicky, H. Leblanc, Galois lattice as a framework to specify algorithms building class hierarchies, Theoretical Informatics and Applications 34 (2000) 521–548.
[42] R.K. Keller, R. Schauer, S. Robitaille, P. Pagé, Pattern-based reverse engineering of design components, in: Proceedings of ICSE '99 (21st International Conference on Software Engineering), IEEE Computer Society Press/ACM Press, May 1999, pp. 226–235.
[43] E.J. Klimas, S. Skublics, D.A. Thomas, Smalltalk with Style, Prentice-Hall, 1996.
[44] C. Kramer, L. Prechelt, Design recovery by automated search for structural design patterns in object-oriented software, in: Proceedings of WCRE '96 (3rd Working Conference on Reverse Engineering), IEEE Computer Society Press, November 1996, pp. 208–216.
[45] S. Kuznetsov, S. Obëdkov, Comparing performance of algorithms for generating concept lattices, in: Proceedings of International Workshop on Concept Lattice-based Theory, Methods and Tools for Knowledge Discovery in Databases, 2001.
[46] W. LaLonde, J. Pugh, Subclassing ≠ subtyping ≠ is-a, Journal of Object-Oriented Programming 3 (5) (1991) 57–62.
[47] D. Lange, Y. Nakamura, Interactive visualization of design patterns can help in framework understanding, in: Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), New York, NY. ACM Press, 1995, pp. 342–357.
[48] M. Lanza, S. Ducasse, A categorization of classes based on the visualization of their internal structure: the class blueprint, in: Proceedings of 16th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01), ACM Press, 2001, pp. 300–311.
[49] M. Lanza, S. Ducasse, L. Steiger, Understanding software evolution using a flexible query engine, in: Proceedings of the Workshop on Formal Foundations of Software Evolution, 2001.
[50] H. Leblanc, Sous-hiérarchies de Galois: un Modèle pour la Construction et L'évolution des Hiérarchies d'objets (Galois Sub-hierarchies: a Model for Construction and Evolution of Object Hierarchies). PhD thesis, Université Montpellier 2, 2000.
[51] A. Lienhard, S. Ducasse, G. Arévalo, Identifying traits with formal concept analysis, in: Proceedings of 20th Conference on Automated Software Engineering (ASE'05), IEEE Computer Society, November 2005, pp. 66–75.
[52] B. Liskov, J.M. Wing. A new definition of the subtype relation, in: O. Nierstrasz (Ed.), Proceedings ECOOP '93, LNCS, vol. 707, Kaiserslautern, Germany, Springer-Verlag, July 1993, pp. 118–141.
[53] A. Mendelzon, J. Sametinger, Reverse engineering by visualizing and querying, Software – Concepts and Tools 16 (1995) 170–182.
[54] P.F. Mihancea, Towards a client driven characterization of class hierarchies, in: Proceedings of International Conference on Program Comprehension (ICPC 2006), Los Alamitos CA. IEEE Computer Society Press, 2006, pp. 285–294.
[55] P.F. Mihancea, R. Marinescu, Discovering comprehension pitfalls in class hierarchies, in: CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, Washington, DC, USA. IEEE Computer Society, 2009, pp. 7–16.
[56] L. Mikhajlov, E. Sekerinski, A study of the fragile base class problem, in: Proceedings of ECOOP'98 (European Conference on Object-Oriented Programming), number 1445 in Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 355–383.
[57] I. Moore, Automatic inheritance hierarchy restructuring and method refactoring, in: Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications), ACM Press, 1996, pp. 235–250.
[58] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: Proceedings of ICSE '02 (24th International Conference on Software Engineering). ACM Press, 2002, pp. 338–348.
[59] J. Niere, J.P. Wadsack, L. Wendehals, Design pattern recovery based on source code analysis with fuzzy logic. tr-ri-01-222, Software Engineering Group, Department of Mathematics and Computer Science, University of Paderborn, Paderborn, Germany, 2001.
[60] O. Nierstrasz, S. Ducasse, Moose – a language-independent reengineering environment, European Research Consortium for Informatics and Mathematics (ERCIM) News 58 (July) (2004) 24–25.
[61] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), New York NY, ACM Press, 2005, pp. 1–10 (invited paper).
[62] I. Philippow, D. Streitferdt, M. Riebisch, Design pattern recovery in architectures for supporting product line development and application, in: ECOOP Workshop–Modeling Variability for Object-Oriented Product Lines. BoD GmbH, 2003, pp. 42–57.
[63] C. Roume, Evaluatioń Structurelle de la Factorisation et la Généralisation au sein des Hiérarchies de Classes: Introduction de Métriques, L'Objet 8 (1-2) (2002) 151–166.
[64] N. Schärli, Traits – Composing Classes from Behavioral Building Blocks. PhD thesis, University of Bern, February 2005.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010

[65] R. Schauer, S. Robitaille, F. Martel, R. Keller, Hot-spot recovery in object-oriented software with inheritance and composition template methods, in: Proceedings of ICSM '99 (International Conference on Software Maintenance). IEEE Computer Society Press, 1999.
[66] J. Seemann, J.W. von Gudenberg, Pattern-based design recovery of java software, in: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, 1998, pp. 10–16.
[67] A. Sharp, Smalltalk by Example, McGraw-Hill, 1997.
[68] F. Shull, W.L. Melo, V.R. Basili, An inductive method for discovering design patterns from object-oriented software systems. Technical Report CS-TR-3597, University of Maryland Computer Science Department, 1996.
[69] G. Snelting, Concept analysis – a new framework for program understanding, in: SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), Montreal, Canada, ACM Press, June 1998, pp. 1–10.
[70] G. Snelting, F. Tip, Reengineering class hierarchies using concept analysis, in: ACM Transactions on Programming Languages and Systems, 1998.
[71] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, vol. 21, Nov. 1986, pp. 38–45.
[72] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, Reuse contracts: managing the evolution of reusable assets, in: Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications). ACM Press, 1996, pp. 268–285.
[73] M. Streckenbach, G. Snelting, Refactoring class hierarchies with KABA, in OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press, 2004, pp. 315–330.
[74] D. Taenzer, M. Ganti, S. Podar, Problems in object-oriented software reuse, in: S. Cook (Ed.), Proceedings ECOOP '89, Nottingham. Cambridge University Press, July 1989, pp. 25–38.
[75] A. Taivalsaari, On the notion of inheritance, ACM Computing Surveys 28 (3) (1996) 438–479.
[76] L. Wendehals, Improving design pattern instance recognition by dynamic analysis, in: Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA), May 2003.
[77] N. Wilde, R. Huitt, Maintenance support for object-oriented programs, IEEE Transactions on Software Engineering SE-18 (12) (1992) 1038–1044.
[78] R. Wuyts, Declarative reasoning about the structure object-oriented systems, in: Proceedings of the TOOLS USA '98 Conference, IEEE Computer Society Press, 1998, pp. 112–124.

Please cite this article in press as: G. Arévalo et al., Generating a catalog of unanticipated schemas in class hierarchies using Formal Concept Analysis, Inform. Softw. Technol. (2010), doi:10.1016/j.infsof.2010.05.010