Mining Source Code for Structural Regularities

Angela Lozano*, Andy Kellens†, Kim Mens*, Gabriela Arevalo‡§
* ICTEAM — Université catholique de Louvain
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium
Email: {angela.lozano | kim.mens}@uclouvain.be
† Software Languages Lab — Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, Belgium
Email: akellens@vub.ac.be
‡ Facultad de Ingeniería — Universidad Austral
Av. Juan de Garay 125 (1063), Buenos Aires, Argentina
§ CONICET
Av. Rivadavia 1917 (1033), Buenos Aires, Argentina
Email: garevalo@austral.edu.ar

***Abstract*—During software development, design rules and contracts in the source code are often encoded through regularities, such as API usage protocols, coding idioms and naming conventions. The structural regularities that govern a program can aid in comprehension and maintenance of the application, but are often implicit or undocumented. Tool support for extracting these regularities from the source code can provide developers useful insights. But building such tool support is not trivial, in particular, because the informal nature of regularities results in frequent deviations and exceptions to these regularities.**

We propose an automated approach, based on association rule mining, to discover the structural regularities that govern the source code of a software system. We chose this technique because of its resilience to exceptions. In general, tool support for mining regularities tends to discover a huge amount of rules, making interpretation of the results hard and time-consuming. To ease the interpretation, we reduce the results to a minimal canonical form, and group them to obtain a more rational description of the discovered regularities. As an initial feasibility study of our approach, we applied it on two open-source systems, namely IntensiVE (Smalltalk) and FreeCol (Java).

I. Introduction

Understanding the rationale and design knowledge of a software system is essential when maintaining or evolving the system [1]. Design decisions and contracts between program entities are often encoded through structural source code regularities, such as naming conventions, design patterns, coding idioms and protocols. However, these regularities are usually not documented explicitly and are frequently violated throughout the source code.

Given that such regularities can reveal where and how important design decisions are implemented, and that deviations of those regularities may hint on problems in the code [2], there have been several approaches to detect, document and track such regularities [3], [4], [5], [6], [7]. However, manual interpretation of discovered regularities can be time consuming, because even when the resulting set of source code entities perfectly matches the regularity, there is little support to infer the design rationale that explains the discovered regularity. We believe that for an automated tool to be successful in extracting structural source code regularities, it should exhibit at least the following properties:

- **Intensional representation:** Rather than presenting the user with a set of source code entities that exhibit the supposed regularity, the intended technique should provide the user with enough information to allow him or her to infer and codify the underlying intent;
- **Robustness towards deviations:** Because deviations to regularities are common, the technique should be sufficiently robust towards deviations, and still be able to detect regularities in the presence of such deviations;
- **Conciseness of results:** Because redundancy in the mined results hampers interpretation of the resulting regularities, an automated tool should minimize the total amount of information presented to the user and maximize the quality of that information.

The main contribution of this paper is a generic lightweight regularity mining approach based on association rule mining that has the above properties and that provides relevant hints to help users in rapidly inferring and documenting the relevance and rationale of potential regularities. Examples of regularities found by our approach include naming conventions, idioms, implementation protocols, design pattern constraints and implementation improvements.

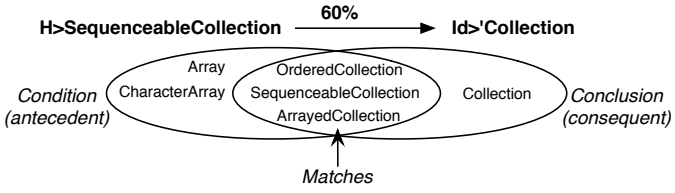
The paper is organized as follows. In Section II we introduce the technique of association rule mining that underlies our approach. Section III describes how we use association rules to locate structural source code regularities. We also describe the various proposed filters and the grouping mechanism aimed at condensing the amount of information that needs to be processed by a user of our approach. In Section IV we describe the two case studies that we have performed to provide a qualitative assessment of our approach. Section V analyses the results of these case studies and takes a more in-depth look at the kinds of regularities we were able to discover. We conclude the paper in Section VII after comparing our work with the state of the art in Section VI.

	H>Collection	H>SequenceableCollection	H>ArrayedCollection	Id>'Collection'	Id>'Array'
Collection	X			X	
SequenceableCollection	X	X		X	
OrderedCollection	X	X		X	
ArrayedCollection	X	X	X	X	X
Array	X	X	X		X
CharacterArray	X	X	X		X

II. Implication and Association rules

This section explains the technique of association rule mining which underlies our approach. Intuitively, association rules can be seen as *if-then* rules such as "if a class belongs to the hierarchy of `AbstractAction`, then it (probably) has the identifier '`Action`' in its name". To illustrate the technique, the table above lists part of the Smalltalk collection classes, together with some properties of those classes, such as inheritance and naming. E.g., `Array` belongs to the class hierarchies of `Collection` and `SequenceableCollection` and has an identifier 'Collection' as part of its name.

An *association rule* is a rule of the form $condition \xrightarrow{confidence\%} conclusion$ that summarizes the extent to which two sets of properties overlap. Intuitively, it can be read as: if an entity satisfies the property $condition$ (the antecedent), then there is a likelihood $confidence$ that it also satisfies property $conclusion$ (the consequent). For example, from the table above we can derive the association rule H>SequenceableCollection $\xrightarrow{60\%}$ Id>'Collection', because from the five subclasses of `SequenceableCollection` that are listed, three also have the identifier 'Collection' in their name. The figure below provides a schematic overview of the structure of an association rule. The set of all entities that satisfy both condition and conclusion (i.e. the intersection of both sets) are called the $matches$ (a.k.a. support set) of the association rule.



The $confidence$ of an association rule represents the percentage of entities satisfying the condition that are actual matches: $confidence = \frac{|matches|}{|condition|}$. However, $confidence$ does not convey information regarding the entities in the conclusion. We define $error$ as a measurement of how many entities in either condition or conclusion are not matches:

$$error = 1 - \frac{\frac{|matches|}{|condition|} + \frac{|matches|}{|conclusion|}}{2}$$

Rules with a high error apply only to a small subset of the involved entities. The closer the error is to zero, the stronger the rule is. Applied to our example rule, we obtain an $error$ of 0.325.

If an association rule has a confidence of 100%, we call it an *implication rule* and denote it with $\Rightarrow$, to distinguish it from weaker association rules. We do not indicate the confidence on implication arrows because it is always 100%. From the table we can derive the implication Id>'Array' $\Rightarrow$ H>ArrayedCollection, which exemplifies an interesting code regularity that can be discovered by an implication and association rule mining approach like ours: all classes with 'Array' in their name belong to the `ArrayedCollection` hierarchy.

III. Our mining approach

Our mining technique takes as input a set of source code entities (classes, methods) along with a number of relations between those entities (inheritance, implementation, naming). The output is a set of association rules that express interesting regularities in the source code.

Our approach consists of the following steps:

1) Determining the kind of input we give to our tool;
2) Pre-filtering of input data to prune redundant and trivial information;
3) Applying the association rule mining algorithm in order to obtain a set of implication and association rules;
4) Post-filtering (simplification and pruning) of resulting rules based on general structural characteristics of these rules and domain-specific heuristics;
5) Grouping sets of related association rules in order to convey the mined results in a more concise manner.

Below we discuss each of these steps in more detail.

A. Input parameters for the algorithm

Although our approach is independent of the actual kinds of source code entities and properties of those entities; in this paper, we restrict our analysis to classes only, to minimize the amount of information to analyze. Thus, we consider the following properties of classes:

- **Hierarchy**: The hierarchy (superclasses) to which a class belongs;
- **Implements**: The names of the methods that are implemented by a class;
- **Identifiers**: The words contained in the name of a class.

We chose these three properties only, because our first analysis discovered relevant relations or correlations only between class hierarchies, implementation protocols and the vocabulary of the system being analyzed.[1] Part of that vocabulary is present in the names of the classes in the system. The class names are split up based on capitalization or delimiting characters to extract the identifiers that belong to the system vocabulary. For example, the class `TextWindowDialog` yields

[1] We also experimented with calling relationships between (methods belonging to) classes and 'refers to' relationships, but discarded those relationships as they did not produce many useful regularities at the level of classes.

identifiers 'Text', 'Window' and 'Dialog' and `Text_Window` yields 'Text' and 'Window').

B. Pre-filtering

The set of properties that serve as input to the association rules algorithm are pruned, before the execution of the algorithm, to remove redundant or irrelevant data. Concretely, we discern two groups of pre-filters:

- **Root classes:** When calculating the *Hierarchy* property of a class, all root classes of the system (e.g. `Object`, the root classes of common libraries and frameworks) are not considered. Including these classes in the analysis results in the deduction of trivial rules that express knowledge such as "all classes are in the hierarchy of `Object`".
- **Common words:** Common words in identifiers ('the', 'get','in',...) are considered meaningless and are pruned.

C. Computation of association rules

Association rules can be computed automatically from a data set [8]. Various algorithms and techniques for mining association rules exist [9]. Let us provide a brief overview of the algorithm we implemented.

Historically, the concepts of association and implication rules stem from Frequent Itemset Analysis. The best-known algorithm for computing such Frequent Itemsets is *Apriori* [8]. For each possible combination of properties, this algorithm calculates the set of entities in the data set that exhibit these properties. Later on, these so-called frequent item sets serve as input to deduce the actual association rules. Calculating the frequent item sets is computationally intensive: all 2^N - 1 (with N being the number of properties) potential association rules are considered by the technique. Although this algorithm computes all possible association rules, its exponential complexity makes it rather impractical.

Our implementation of an association rule miner avoids this combinatorial explosion by restricting the sets of properties between which we compute possible association rules. More specifically, we calculate association and implication rules based on a concept lattice. To this end, we employ an efficient Formal Concept Analysis (FCA) algorithm. FCA is a branch of lattice theory that allows identifying meaningful and maximal groupings of elements that share a set of common properties [10]. Our implementation of FCA has a runtime complexity of $O(N^2)$, with N being the number of properties. In the resulting lattice, we obtain groups of properties that belong together, according to the data set. By only considering the concepts that introduce a property (maximal N concepts), and calculating the association rules between that concept and the other concepts on its path, we need only consider N^2 different combinations of properties for the association rules.

D. Post-filtering

To reduce the amount of information presented to the user, we also post-process and post-filter the association rules reported by our algorithm. We distinguish two kinds of filters, namely *structural filters* that aim at rewriting the mined association rules and removing redundancy from these rules and *heuristic filters* that use domain knowledge to further restrict the amount of (meaningless) results. Table I shows an overview of all filters and their definition. Below, we take a more in-depth look at each of these filters, and describe their workings.

1) Structural filters: A first group of filters are the structural filters. These filters are based on structural properties of the mined association rules and aim both at eliminating irrelevant rules (rules subsumed by other rules), and making the rules more concise. These filters are oblivious to the actual properties of the dataset being analyzed and can therefore be reused when applying our approach using different input properties. In this category, we have defined the following eight rules[2]:

- **Matches (S1):** Rules that have 4 or less matches are considered not to convey meaningful information, as these association rules are not supported by a sufficiently large part of the data set. In general, less than 3 matches cannot be considered a regularity (1 is a singularity, 2 is a casualty, and from 3 matches and on it can be considered a regularity). We chose a threshold of 4 matches for our case studies, because regularities of just 3 matches were only adding noise.
- **Left to Right (S2):** This filter favors association rules with higher confidence. If we have a rule $A \rightarrow B$ and a rule $B \rightarrow A$, then we eliminate the rule with lowest confidence.
- **Confidence (S3):** This filter prunes rules with confidence less than 70%. Intuitively, if a rule is valid for less than 70% of the elements satisfying the condition of the rule, we do not consider it to be an interesting regularity because there is too much counter-evidence.
- **Error (S4):** Rules with a degree of error larger than 45% are discarded as well. Intuitively, even when a rule is supported by a certain number of elements in the data set (the matches), we discard it when there are, in comparison, many exceptions in either the condition (low confidence) or the conclusion (low coverage) with respect to the matches.
- **Compact Rules (S5):** This filter aims at removing redundant information from the condition or conclusion of an association, by using the information contained in the implications. An implication represents a subset relation between two sets of properties: if an entity satisfies the properties in the condition of the implication, then it will also satisfy the properties in the conclusion. We can use this information to remove redundancies in the set of computed association rules. If the condition or conclusion of an association rule contains *both* condition and conclusion of the implication, then we can safely remove the property corresponding to the conclusion of the implication from the association rule without any

[2]For those filters that rely on a certain threshold value, those thresholds were experimentally determined based on the two case studies conducted, but may need to be fine-tuned for future case studies.

Filter name (id)	Before filtering	After filtering (rules that remain i.e. the more specific rules)
Matches (S1)	$R : A \to B$	$\begin{cases} A \to B & \texttt{if } \lvert matches(R)\rvert \geq 4 \\ discard & \texttt{otherwise} \end{cases}$
Left to right (S2)	$R_1 : A \to B$ $R_2 : B \to A$	$\begin{cases} A \to B & \texttt{if } \frac{\lvert matches(R_1)\rvert}{\lvert A\rvert} > \frac{\lvert matches(R_1)\rvert}{\lvert B\rvert} \\ B \to A & \texttt{otherwise} \end{cases}$
Confidence (S3)	$R : A \to B$	$\begin{cases} A \to B & \texttt{if } confidence(R) \geq 70\% \\ discard & \texttt{otherwise} \end{cases}$
Error (S4)	$R : A \to B$	$\begin{cases} A \to B & \texttt{if } error(R) \leq 45\% \\ discard & \texttt{otherwise} \end{cases}$
Compact Rules (applying implications) (S5)	$A \wedge B \to C$ $E \to F \wedge G$ $A \Rightarrow B$ $F \Rightarrow G$	$A \to C$ $E \to F$ $A \Rightarrow B$ $F \Rightarrow G$
Redundant mention of super-property (S6)	$A \to B \wedge C$ $A \Rightarrow B$	$A \to C$ $A \Rightarrow B$
Super & sub-properties are concluded by same properties (S7)	$R_1 : X \to A$ $R_2 : X \to B$ $A \Rightarrow B$	$\begin{cases} X \to A & \texttt{if } confidence(R_1) \geq confidence(R_2) \\ & \quad \wedge error(R_1) \leq error(R_2) \\ X \to B & \texttt{otherwise} \end{cases}$ $A \Rightarrow B$
Super & sub-properties conclude same properties (S8)	$R_1 : A \to X$ $R_2 : B \to X$ $A \Rightarrow B$	$\begin{cases} A \to X & \texttt{if } confidence(R_1) \geq confidence(R_2) \\ & \quad \wedge error(R_1) \leq error(R_2) \\ B \to X & \texttt{otherwise} \end{cases}$ $A \Rightarrow B$
Hierarchy relations (H1)	$H > Subclass \Rightarrow H > Superclass$	$discard$ if $Subclass$ is subclass of $Superclass$
Semantic pruning (H2)	$A \to B$	$\begin{cases} A \to B & \texttt{if } weight(A) > weight(B) \\ discard & \texttt{otherwise} \end{cases}$ with: $weight(Hierarchy) > weight(Implements)$ $weight(Hierarchy) > weight(Identifier)$ $weight(Implements) = weight(Identifier)$

TABLE I
OVERVIEW OF THE POST-FILTERS

impact on the amount of information conveyed in the rule. For example, if we have an association rule $A \wedge B \to C$ and an implication $A \Rightarrow B$, then we can simplify this association rule to $A \to C$ (note that A is more specific). Note that multiple implications can possibly be applied to a single association rule. In such cases, the order in which the implications are applied can have an impact on the outcome of this filter, if there exists an overlap between the properties of the implications. To circumvent this problem, our approach sorts the applicable implications such that a maximum of implications can be applied to a single association rule. This is achieved by ordering the implications such that those ones whose conclusion is the condition of another implication that is applicable get preference. To illustrate this, consider an association rule $A \wedge B \wedge C \to D$ and two (applicable) implications $A \Rightarrow B$ and $B \Rightarrow C$. If we start by applying the first implication, this will remove B from the condition of the association rule, making it impossible to apply the second implication. However, the other way around we can apply both implications, resulting in the association rule $A \to D$.

- **Redundant mention of super property (S6):** This filter is based on the same principles as **S5**, and aims to improve the conciseness of mined implications. In contrast to **S5**, this filter allows the conclusion and the condition of the implication to be in different parts of the association rule. In fact, the condition of the implication must be part of the condition of the association, and the conclusion of the implication must be part of the conclusion of the association. The converse does not apply, because applying the implication would change the meaning of the rule. For instance, $B \wedge C \to A$, $A \Rightarrow B$ cannot be simplified to $C \to A$, $A \Rightarrow B$ because the association rules do not permit to conclude that C is a more specific expression of $B \wedge C$. On the contrary, the association rules presented in Table I permit to conclude that A is a more specific expression of $A \wedge B$.
- **Super & sub-properties concluded by same properties (S7):** The goal of this filter is to remove a redundancy in the mined information in situations where we have two association rules $X \to A$ and $X \to B$, where there also exists the implication $A \Rightarrow B$. To decide which rules gets pruned, we compute the $confidence$ and $error$ of both rules and remove the one with lowest $confidence$ and highest $error$. This filter is based on the observation that, although A is a subset of B, this does not necessarily imply that there exists more overlap between X and B

than between X and A.

- **Super & sub-properties conclude same properties (S8):** This filter is complementary to **S7** discussed above. This filter prunes, according to the $confidence$ and $error$, the redundant rule in the situation where we have association rules $A \rightarrow X$ and $B \rightarrow X$, and where $A \Rightarrow B$.

2) Heuristic filters: The above set of general filters are complemented by two heuristic filters. These filters are dedicated to the experimental setup of this paper and rely on the semantics of the different kinds of analyzed properties in order to further restrict the set of mined association rules.

- **Hierarchy relations (H1):** One of the properties of the classes we analyze is the class hierarchy relationship. This causes our algorithm to identify implication rules that express mere subclass relationships (e.g., "if a class is in the hierarchy of class `A`, it also is in the hierarchy of class `B`" is trivially true for each superclass `B` of `A`). Since such implication rules present trivial information, they are pruned from the final result.
- **Semantic pruning (H2):** Experimentation with earlier versions of our approach revealed that in many cases, some association rules were deemed to be less interesting in the context of mining structural regularities than others. In our case studies with classes, rules whose conditions only contained identifier properties, or whose condition contained implemented messages and the conclusion a particular hierarchy, were observed to be of less interest to the developer. To prune away such rules, we defined a partial order between the properties under consideration and pruned away rules where the condition appeared to be of less interest than the conclusion of the rule.

E. Grouping

Upon manual inspection of the association rules reported by our tool we observed that the number of association rules can be considerably large. However, in many cases, several association rules describe properties of the same set of source code entities, and are often contributing to the same regularity. Therefore, we further restrict the amount of information conveyed to the user by grouping such related rules. In practice, we consider two association rules to belong to the same group (i.e., contributing to the same structural regularity) if at least 70% of their $matches$ overlap.

F. Discussion

Our technique for identifying meaningful structural regularities in the source code of programs, has the desirable characteristics mentioned in Section I:

- *It provides useful clues on the rationale behind potential regularities*, by merging all shared properties for related source code entities: the more information we have on what properties the entities have in common, the more clues we have on why these entities are related.
- *It provides an intensional description of the regularities.* Rather than just mining for sets of entities that share some regularity, we also discover their intension, i.e., the properties and rules those entities have in common.
- *It allows for deviations.* By using a mining technique (association rules) that does not require exact matches of the regularities, we increase the tolerance for deviations. At the same time, by tweaking the tolerance level so that potential regularities with too many mismatches get discarded, we can discard easily regularities that are likely to be false positives.
- *It provides concise results* by eliminating similar regularities, simplifying regularities to their minimal expression, and grouping regularities that cover the same sets of entities.

IV. Case studies

	IntensiVE	**FreeCol**
Implementation language	Smalltalk	Java
Number of classes	233	382
Number of methods	2863	3252
Lines of code	12318	31191

TABLE II
OVERVIEW OF THE TWO CASE STUDIES

As an exploratory study of the usefulness of our approach and to perform a qualitative assessment of the mined results, we applied it to two open-source applications. More specifically, we studied the `IntensiVE` and `FreeCol` systems. Table II provides an overview of the size of both systems.

- **IntensiVE** is an academic tool suite that supports the documentation and verification of structural source code regularities. It was written (amongst others) by the second and third author of this paper. Next to supporting regularities, its implementation relies heavily on the use of regularities. Since IntensiVE's regularities are well-documented and we possess expert knowledge about these regularities, IntensiVE was an ideal first case study on which to test our approach.
- **FreeCol** is a free and open implementation of the game 'Colonization'. In contrast to the other case study, we chose this system since it was new to all authors of the paper, however presents a well-known domain.

Table III gives an overview of the size of the mined results, and how our filters were able to reduce the amount of information conveyed to the user. Despite the fact that our approach uses Formal Concept Analysis to compute the association rules (and does not perform a brute force combination of all properties), before filtering, our algorithm still produced an overwhelming amount of association rules (approximately 75,000 for IntensiVE, and over 200,000 for FreeCol). As can be seen in the table, the vast majority of these rules got pruned by filter S1, which removes all rules that have less than 4 matches and are therefore considered to be irrelevant. Filter S3 (rules with too low confidence) further prunes the set of rules for both case studies by approximately 2000 rules. The remaining filters reduce the set of rules further by

	IntensiVE	FreeCol
Execution time	47 seconds	3 minutes
Inspection time	2 hours	8 hours
Rules before filtering	75823	203305
Pruned by Filter S1	72814	200760
Pruned by Filter S2	17	15
Pruned by Filter S3	2279	1905
Pruned by Filter S4	1	7
Pruned by Filter S5	39	17
Pruned by Filter S6	1	0
Pruned by Filter S7	32	22
Pruned by Filter S8	39	16
Pruned by Filter H1	12	25
Pruned by Filter H2	6	6
Duplicate rules	437	417
Rules after filtering	146	115
Groups	38	27

TABLE III
OVERVIEW OF THE RESULTS OF APPLYING OUR TECHNIQUE TO THE CASE STUDIES.

approximately 150 rules (IntensiVE) and 100 rules (FreeCol). Note that these filters not only remove redundant rules, but that they (e.g. in the case of S5) reduce rules to a more concise form. After removing duplicate rules[3], we end up with 146 rules for IntensiVE and 115 rules for FreeCol. After grouping, we end up with 38 groups for IntensiVE and 27 for FreeCol.

Table III also includes the time necessary to compute the set of association rules as well as the time invested by the authors of the paper in analyzing the resulting groups. Our approach is reasonably fast in computing the rules, ranging from less than a minute for IntensiVE to approximately 3 minutes for FreeCol. The time necessary to inspect the resulting groups differs drastically between both case studies. In the case of IntensiVE, it took about 2 hours to go over each of the proposed groups, analyze the association rules in that group and inspect the elements that matched the rules, along with possible deviations of the rules. As we knew the implementation of IntensiVE well, the amount of needed time for performing such a detailed analysis was fairly restricted. For FreeCol, such an analysis took about 8 hours. This was not unexpected as we did not know the implementation details of that system and therefore had to invest a larger effort to assess the groups and association rules proposed by our tool.

V. ANALYSIS OF RESULTS

In this section, we take a look at the various kinds of regularities that our approach was able to identify in the two case studies. Despite the fact that this paper presents only an initial exploratory study of the use of association rule mining to identify structural source code regularities, and therefore only considered three simple properties (class identifiers, class hierarchies, methods implemented), our approach was able to identify a number of interesting regularities.

We distinguish three kinds of identified regularities:

- **Naming conventions:** Regularities that express how the classes in a particular hierarchy, or implementing a particular concern or protocol, should be named.
- **Complementary methods:** Regularities that express sets of methods that should be implemented together.
- **Interface definitions:** Regularities that express the set of methods that have to be implemented when a class is present in a particular hierarchy, or when it implements a particular concept.

In what follows, we discuss each of these kinds of regularities in more detail and provide examples of interesting identified regularities from the two case studies. Within these examples, we use the following naming conventions for indicating properties: identifiers are indicated by 'Id', class hierarchies by 'H' and implements relationships by 'Im'. For example, the property that a class belongs to the hierarchy of `Object` is indicated by $H > Object$.

A. Naming conventions

Our approach identified a number of association rules, or groups of rules representing naming conventions. These association rules are characterized by the fact that their conclusion contains one or more 'Id' properties, and represent regularities such as "If a class belongs to a hierarchy C, it should have identifier K and L in its class name", or rules such as "If a class implements a method named M, then that class should have identifier K in its name".

1) IntensiVE: The following two examples are selected from the naming conventions found in IntensiVE:

a) Quantifiers: IntensiVE offers developers a set-theoretic model to document structural source code regularities. Part of this model (and the corresponding implementation) defines logic quantifiers such as $\forall, \exists$, and so on. Each kind of quantifier is implemented by a separate class (in the hierarchy of `AbstractQuantifier`) that follows the naming convention that its name contains the identifier '`Quantifier`'. Our approach identified a group of three implications that represent this regularity:

$$Im > symbol \Rightarrow Id >' Quantifier' \wedge H > AbstractQuantifier$$
$$Id >' Quantifier' \Rightarrow Im > symbol \wedge H > AbstractQuantifier$$
$$H > AbstractQuantifier \Rightarrow Im > symbol \wedge Id >' Quantifier'$$

These rules encode the knowledge that *all* classes in the hierarchy of `AbstractQuantifier` implement the method named `symbol` (representing the mathematical symbol for the quantifier) and have '`Quantifier`' as part of their class name. As the identified rules were implications, there are no exceptions to these rules: all 14 quantifier classes in IntensiVE obey the regularity.

b) Visualizations: IntensiVE features a visual query language. For each of the visual primitives in this query language (e.g. classes, methods, associations, ...) a separate class is used. All of these classes are characterized by the fact that they should contain the identifiers 'IVEditor' and 'Figure'. A group of three association rules that was identified by our tool

[3]Duplicate rules may be introduced when rules are compacted by filters like S5. For example, a more complex rule may be compacted to a more simple rule that already existed.

represents this naming convention:

$$Id >' IVEditor' \xrightarrow{96\%} Id >' Figure'$$
$$H > IVEditorFigure \xrightarrow{95\%} Id >' IVEditor'$$
$$H > IVEditorFigure \xrightarrow{95\%} Id >' IVEditor' \wedge Id >' Figure'$$

The first rule expresses that if a class contains the identifier 'IVEditor' it most likely contains 'Figure' as well. The other rules express that classes in the hierarchy of `IVEditorFigure` should follow the naming convention. Note that, while these three rules have a high confidence, there exists a single exception to these rules. Further inspection of the rules revealed that one class, `IVEFigureRootCollection`, did not obey the regularity and was therefore corrected to `IVEditorFigureRootCollection`.

2) FreeCol: Similar to the IntensiVE case study, we identified a number of interesting naming conventions within FreeCol. One example is the following group:

$$Id >' Action' \wedge Im > getID \Rightarrow H > FreeColAction$$
$$H > FreeColAction \Rightarrow Id >' Action'$$
$$H > FreeColAction \wedge Im > actionPerformed \Rightarrow Id >' Action'$$
$$H > FreeColAction \wedge Im > shouldBeEnabled \Rightarrow Id >' Action'$$

FreeCol implements a hierarchy of classes that represent user interface actions. In addition to the fact that all of these actions belong to the hierarchy `FreeColAction`, they all share the naming convention that they contain the identifier 'Action' in their class name. Note that this group does *not* include a rule that concludes the hierarchy from the identifier, as this is not the case in the source code. For example, there exist classes like `ActionManager` that — while they contains the proper identifier — do not implement a user interface action.

B. Complementary methods

The second kind of regularity that was identified by our approach are groups of methods that all contribute to the implementation of a particular concept. These rules represent regularities of the form "If a class implements a method named M, then it should also implement a method named N".

1) IntensiVE:

a) Compilation: One instance of this kind of regularity that we identified in IntensiVE was the saving mechanism. Crosscutting the entire implementation of IntensiVE, there are classes that represent objects that can be compiled. To achieve this, IntensiVE offers a small framework that is customized by each compilable object. Our approach identified a fairly large group of 21 implication rules that demonstrate the use of this framework. Some of the rules that were part of this group are:

$$Im > compileFooterOn :\Rightarrow \quad Im > compileDefinitionOn :$$
$$Im > compileCachingOn :\Rightarrow \quad Im > save\wedge$$
$$Im > compileHeaderOn : \wedge$$
$$Im > compileFooterOn : \wedge$$
$$Im > compileSpecificsOn :$$
$$Im > fullname \wedge Im > save \Rightarrow \quad Im > generateSelector$$
$$\dots$$

This regularity describes the set of methods that need to be overridden by an object in order to be compilable. If a developer overrides one of these methods, then most likely all others should also be overridden.

b) Undo: The actions that can be performed within the user interface of IntensiVE are implemented by means of a Command design pattern [11]. A subset of these actions are undoable. This is indicated in the source code by the fact that these actions implement the method `isUndoable`. However, in such cases, the undoable action should also implement the method `undoAction` that performs the actual undo. Furthermore, undoable actions are allowed to implement the optional `redoAction` method in order to perform a redo.

Our approach identified a group with two rules hinting at the above regularity:

$$Im > isUndoable \Rightarrow Im > undoAction$$
$$Im > redoAction \Rightarrow Im > name, Im > undoAction$$

2) FreeCol: Within FreeCol, we identified a group that contains a single association rule. In particular, this group expressed that:

$$Im > readFromXMLImpl \xrightarrow{96\%} Im > getXMLElementTagName$$

A closer inspection of the source code identified that FreeCol offers serialization of game objects to XML files. In order to properly work, an object needs to implement both the `readFromXMLImpl` and `getXMLElementTagName` methods. There were two exceptions to this rule, which represent abstract classes that implement `readFromXMLImpl`, but that require their concrete subclasses to specify `getXMLElementTagName`. Note that this regularity could not be captured by a Java interface: it does not suffice for a class to inherit an implementation of `getXMLElementTagName`; each class that implements `readFromXMLImpl` should implement its *own* `getXMLElementTagName` method.

C. Interface definitions

The third kind of regularity that we identified are interface definitions. These are represented by association rules of the form "If a class belongs to hierarchy C, then it should implement methods named M and N".

1) IntensiVE: IntensiVE is implemented in the dynamic language Smalltalk, which does not offer the language construct of an interface. As such, some of the best-documented regularities in IntensiVE are interface definitions. Our approach was able to extract some of these interface definitions automatically from the source code. For example, IntensiVE offers the concept of fuzzy quantifiers. These are quantifiers such as "almost all", "most" and "many". Within the implementation of IntensiVE, these fuzzy quantifiers have to implement a particular interface to indicate that they are a fuzzy quantifier, and to calculate their truth degree. One of the groups of association rules that was identified by our approach contained the following implication rule that expresses this interface definition:

$$H > FuzzyQuantifier \Rightarrow \quad Im > crispQuantifier\wedge$$
$$Im > holdsWithTruth : total\wedge$$
$$Im > fuzzyDegreeWithTruth : total :$$

The above rule describes the exact intent of the regularity in IntensiVE, namely that all classes in the hierarchy of `FuzzyQuantifier` should implement the correct interface.

2) FreeCol: The colonization game contains the concept of *Locations*: the various kinds of tiles in the game (such as colonies, settlements, and so on). Internally in the implementation, this concept is represented by an interface definition named `Location` that specifies that locations need to provide an implementation for methods `getGoodsContainer` and `getLocationName`. While this regularity is therefore explicitly verified by the Java language itself, our approach was able to identify it in the source code. More specifically, we found a group containing amongst others the following rules:

$$H > Location \xrightarrow{80\%} Im > getGoodsContainer$$
$$H > Location \xrightarrow{80\%} Im > getLocationName$$

Notice that this rule has a confidence of only 80% because there exist an abstract class in the implementation of FreeCol that delegates the implementation of the methods to its subclasses. Our approach did not mine the opposite rule (where the implementation of the methods concludes the interface) because of the presence of sub-interfaces of `Location` that require additional methods to be implemented.

D. Discussion

1) Grouping of rules: One of the contributions of our approach is the fact that we propose to group rules of which the matches show a significant amount of overlap. In both case studies, this grouping of rules offered, next to condensing the amount of information that needs to be processed by a developer, the added benefit of allowing the evaluation of association and implication rules in a particular context. Section V-B1a presents an example of this. We discussed the regularity that, in order to correctly compile an entity, that entity's class needs to implement a set of methods. This regularity was described by means of a large number of association rules. If the association rules would have been inspected individually, it would have been easy to miss that all of these rules actually belong together and describe the implementation of a particular concept. In other words, the groups aid in discovering the intent of a particular regularity by grouping all of its properties.

2) Heterogenous versus homogenous groups: Most of the identified groups were heterogenous, meaning that the rules in these groups did not exclusively describe a naming convention, an interface definition or complementary methods. This is not that surprising since our technique groups together rules based on their matches. Consequently, a group reported by our approach does often not align with a single regularity, but rather consists of multiple regularities that are applicable to the same set of entities.

3) Finding violations and improvements using the mined rules: One of the strengths of association rule mining — which motivated our choice for this technique — is its resilience to deviations in the data set. In other words, in order for a regularity to be discovered by our approach, it is not necessary that this regularity is strictly obeyed throughout the entire source code. During our analysis, we found that most of these *imperfect* regularities were caused by the fact that the regularity itself is not always applicable (i.e. there exist exceptions to the general rule), we were also able to identify a number of violations of the regularities. For example, as we discuss in section V-A1b, our approach amongst others identified a violation of one of the naming conventions within IntensiVE. While not discussed above, in the FreeCol case study we identified a refactoring opportunity in the graphical interface where a dedicated subclass of `JPanel` was used. For no apparent reason, some classes in the interface did not use this dedicated class but extended the `JPanel` class directly.

4) Choice of properties and influence on mined regularities: Since it was our goal to demonstrate the feasibility of our approach, for now we restricted our analysis to classes and opted to only include three simple properties of classes. As a consequence, the different kinds of regularities that can be discovered based on these properties is rather limited. We can observe that most of the reported rules describe the vocabulary that is used in the application. While this provides interesting information regarding the application [12] (for example, to novice developers), a wide range of regularities are currently not mined by our approach. For example, calling relationships, usage of particular classes, framework specialization constraints, and so on are not found. We plan further experimentation where we will not restrict our analysis to classes but also include methods. Furthermore, we will investigate the use of properties such as calling relationships, method overrides, typing information and statement ordering.

5) Amount of manual effort: Possible scalability issues of our approach are not caused by the complexity of computing the actual association rules, but rather by the manual analysis of the resulting groups. While our approach aims at minimizing the amount of effort that needs to be invested by a developer (by extensive filtering and grouping of rules), it still requires a considerable effort to extract the rationale behind the grouping. In order to identify the intent of a group, a developer needs to inspect the various rules in that group, along with the source code entities that match the rule, and the exceptions to this rule. Especially in the case of an unfamiliar system, this can be time consuming. Although such a manual analysis has to be performed only once for a particular system, we plan to further circumvent this problem in future work by allowing the mining and analysis to be done in a more incremental way, during the development process. We envision a tool that extends a standard IDE in such a way that, when browsing a particular source code entity, the developer is also shown the groups and association rules in which this source code entity is involved (either as a match to a rule, or as a deviation). This way, a manual post-processing step of the results of our approach is no longer necessary but the analysis can be done incrementally, during development.

6) Correctness of the results: From a technical point of view, all rules mined by our approach are correct with respect to the system that is analyzed. Nevertheless, without the filtering that is offered by our approach, the use of association rule mining can still result in a large number of trivial

rules. Since we restrict our rules to those that are supported by a sufficient number of matches, and that exhibit a high confidence and low degree of error, we are able to prune a significant number of these trivial rules. This however does not provide any insights into the quality of the mined results from a usability perspective: are the groups of rules identified by our tool of interest to developers? While we were able to identify interesting regularities in both case studies, such an analysis of the usefulness of our approach is highly subjective. In future work, we will validate our approach by mining for regularities in a number of open-source systems and involve the original developers of these systems in a user study.

7) Comparison with known regularities in IntensiVE: For the IntensiVE tool, a fairly large number of regularities were already documented. While our approach was not able to find all of these regularities (due to restricted scope of our experiment), we were able to identify a majority (10 out of 15) of the documented naming conventions, complementary methods and interface definitions. For example, the compilation scheme, the undoable actions and the quantifier naming convention discussed above were all documented already using the IntensiVE tool. Furthermore, eight interesting regularities such as the fuzzy quantifier interface and part of the user interface protocol were identified by our approach, but were not previously documented. Our approach however failed to identify the naming conventions and interface definitions related to the implementation of a Factory design pattern within IntensiVE. The reason for this is the fact that, due to the small number of classes that have to respect these regularities, association rules that involved the classes implementing the factory were removed from the result by our pruning filters.

8) Choice of thresholds: Our approach can result in false negatives as consequence of the aggressive filtering scheme where we expect at least 4 matches for the rule, a confidence of 70% and a degree of error lower than 45%. While this filtering strategy allows us to limit the amount of noise, it can result (as illustrated in the previous point) that certain interesting regularities are filtered out. First, our choice for expecting at least 4 matches for a rule was inspired by the observation that a lower threshold would include casual correlations of properties, therefore increasing the amount of noise produced by our approach. Second, we experimented with different confidence and degree of error thresholds. Both these thresholds have an impact on the number of entities that get reported by our tool and were determined after experimentation. For example, using a confidence threshold of 50% on the IntensiVE case study only resulted in 40 more association rules, and 2 extra groups. However, we observed that this lower threshold resulted in the introduction of random association rules in the groups, that were caused by seemingly unrelated properties. Although the confidence of these rules was higher than the threshold, both condition and conclusion exhibited a large number of exceptions. Similar observations can be made for the degree of error. While a lower degree would result in less groups/rules, and a number of interesting regularities would be eliminated, too high a degree of error would allow for rules that are too generic (coincidental). While experiments resulted in the same thresholds for both case studies, this does not imply that these thresholds are optimal for the analysis of any system. We hypothesize that, since the size of both systems is similar, the same thresholds yielded satisfying results (in spite of the fact that the systems were written in different languages).

VI. Related work

A. Approaches using association rule mining

Our work is not the first one to propose the use of association rule mining for extracting knowledge from source code. For example, Michail [13] uses association rules to mine library reuse patterns (library components that are reused together). Similar to our approach, this work analyzes classes and basic relationships between these classes. Furthermore, this approach also proposes some heuristics to filter rules that are not adding information. Thummalapental and Xie [14] mine association rules obtained from exception handling code. The association rules that are obtained by their approach are composed of sequences of method calls that occur together. Bruch et al. [15] propose the use of frequent itemset analysis and association rule analysis to remove irrelevant recommendations in code completion. PR-Miner by Li and Zhou [16] use association rules to identify violations to correlations between function calls. Similar to our approach, they offer a grouping mechanism; theirs is based on the antecedent of the rules.

The above approaches are similar to our approach in that they use the same mining technique and similar input data. However, the nature of the problem tackled differs. While previous approaches aim at finding accurate examples or detecting exceptions, we aim at presenting the mined association rules in a condensed format to ease the interpretation of the rationale behind these rules. Therefore, the filtering and grouping strategies we propose differ from previous ones.

B. Mining for structural regularities

There are several approaches specialized in mining certain types of source code regularities. For instance, aspect mining techniques [17] look for frequent scattering patterns, however these approaches tend to be difficult to interpret and intolerant to exceptions [18]. Another set of approaches has been proposed to detect features (i.e., the implementation of user-perceivable functionality). However, feature detection is generally based on data that is difficult to extract such as vectors of words that characterize source code entities [19], and data and control flow relations [4], [20]. Other feature identification techniques rely on traces [21], [7], [22], which require intense filtering to separate the calls to auxiliary methods from those that indeed implement the feature. Furthermore, there are several approaches to mine for API usages [13], [23], [24], [25] however, the results tend to be very low-level implementation rules, which are difficult to interpret as domain/application specific rules. Finally, there are approaches that aim at extracting domain/application design rules such as [26], nevertheless the scalability of this approach

is limited. To summarize, in comparison to our approach, previous regularity mining approaches tend to be specialized for a single type of regularity, while our approach would find as many and diverse regularities as properties analyzed. In contrast to the approaches presented, ours is a lightweight approach which is resilient to exceptions and provides hints to interpret the rationale of the results.

VII. Conclusions

Structural source code regularities such as idioms, naming conventions, interface definitions, play an important role in easing software maintenance and evolution. Unfortunately, such regularities are often only implicitly known and not documented. Furthermore, automatic extraction of such structural regularities from source code is not a trivial task, due to the overwhelming amount of information that mining techniques might present a user, and the inherent presence of exceptions to the structural regularities in the source code.

In this paper we have presented a novel approach for mining such regularities that is based on association rule mining. The contributions of our approach are:

1) Resilience to exceptions in the source code, due to the nature of the applied association rule mining technique;
2) A comprehensible representation of the mined rules, due to an elaborate post-filtering and grouping of rules;
3) An intentional description of the mined regularities, due to the fact that we mine for relations between *properties* of source code entities, and not between the actual entities.

As a proof-of-concept of the feasibility of our approach, we have applied it to two open-source systems, one in Smalltalk and one in Java. Despite the fact that we only considered three simple properties of the analyzed classes (identifiers, implemented methods, inheritance relationships), our qualitative analysis of the resulting groups of association rules indicated that our approach is able to discover interesting structural source code regularities. Currently, we are extending the experiment to methods and relations between those methods.

Acknowledgements

This work has been performed under the scope of the MinDeR bilateral project sponsored by MINCyT Argentina and FWO Flanders. Angela Lozano is funded by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB). Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This work has also been supported by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State – Belgian Science Policy.

References

[1] K. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *The Future of Software Engineering*, 2000, pp. 73–87.
[2] T. Matsumura, A. Monden, and K. Matsumoto, "The detection of faulty code violating implicit coding rules," in *Workshop on Principles of Software Evolution*, 2002.
[3] K. Gallagher and J. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Softw. Eng.*, vol. 17, no. 8, pp. 751–761, 1991.
[4] K. Chen and V. Rajlich, "Case study of feature location using dependence graph." in *Intl. Workshop on Program Comprehension*, 2000, pp. 241–247.
[5] M. Robillard and G. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *Intl. Conf. on Software Engineering*. ACM, 2002, pp. 406–416.
[6] K. Mens, I. Michiels, and R. Wuyts, "Supporting software development through declaratively codified programming patterns," *Elsevier Journal on Expert Systems with Applications*, vol. 23, no. 4, pp. 405–431, 2002.
[7] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, 2003.
[8] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases." in *Intl. Conf. on Management of Data*. ACM SIGMOD, 1993, pp. 207–216.
[9] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
[10] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, 1995.
[12] J. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Intl. Conf. on Software Engineering*. IEEE/ACM, 2001, pp. 103–112.
[13] A. Michail, "Data mining library reuse patterns using generalized association rules," in *Intl. Conf. on Software Engineering*. ACM, 2000, pp. 167–176.
[14] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *ICSE '09: Proc. of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 496–506.
[15] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *European Software Engineering Conf. and the symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 213–222.
[16] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *European software engineering conference/International symposium on Foundations of software engineering*. ACM, 2005, pp. 306–315.
[17] A. Kellens, K. Mens, and P. Tonella, "A survey of automated code-level aspect mining techniques," *Transactions on Aspect-oriented Development (TAOSD)*, 2007.
[18] K. Mens, A. Kellens, and J. Krinke, "Pitfalls in aspect mining," in *Working Conf. on Reverse Engineering*. IEEE, 2008, pp. 113–122.
[19] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Intl. Conf. on Software Engineering*, 2003, pp. 125–135.
[20] B. Dagenais, S. Breu, F. Warr, and M. Robillard, "Inferring structural patterns for concern traceability in evolving software." in *Automated Software Engineering (ASE)*. ACM, 2007, pp. 254–263.
[21] N. Wilde, M. Buckellew, H. Page, and V. Rajlich, "A case study of feature location in unstructured legacy FORTRAN code." in *European Conf. on Software Maintenance and Reengineering*, 2001, pp. 68–76.
[22] G. Antoniol and Y. Guéhéneuc, "Feature identification: A novel approach and a case study," in *Intl. Conf. on Software Maintenance*, 2005, pp. 357–366.
[23] C. Williams and J. Hollingsworth, "Automatic mining of source code repositories to improve bug funding techniques." *Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.
[24] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories." in *Mining Software Repositories*. ACM, 2006, pp. 54–57.
[25] H. Kagdi, M. Collard, and J. Maletic, "An approach to mining call-usage patterns with syntactic context." in *Intl. Conf. on Automated Software Engineering*, 2007, pp. 457–460.
[26] P. Lam and M. Rinard, "A type system and analysis for the automatic extraction and enforcement of design information." in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 275–302.