Using Concept Analysis to Detect Co-Change Patterns

In Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007)

Tudor Gîrba
Software Composition Group
University of Bern

Stéphane Ducasse
LISTIC
University of Savoie

Adrian Kuhn
Software Composition Group
University of Bern

Radu Marinescu
LOOSE Research Group
Technical University of Timişoara

Raţiu Daniel
Institute for Informatics
Technical University of München

ABSTRACT

Software systems need to change over time to cope with new requirements, and due to design decisions, the changes happen to crosscut the system's structure. Understanding how changes appear in the system can reveal hidden dependencies between different entities of the system. We propose the usage of concept analysis to identify groups of entities that change in the same way and in the same time. We apply our approach at different levels of abstraction (*i.e.,* method, class, package) and we detect fine grained changes (*i.e.,* statements were added in a class, but no method was added there). Concept analysis is a technique that identifies entities that have the same properties, but it requires manual inspection due to the large number of candidates it detects. We propose a heuristic that dramatically eliminate the false positives. We apply our approach on two case studies and we show how we can identify hidden dependencies and detect bad smells.

Keywords: co-change analysis, concept analysis, evolution analysis

1. INTRODUCTION

Software systems need to change over time to cope with the new requirements [14]. However, as requirements happen to crosscut the system's structure, changes will have to be made in multiple places.

Research has been carried out to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes: to identify hidden architectural dependencies [6], to point developers to possible places that need change [21], or to use them as change predictors [11].

The detection is mostly based on mining versioning systems like CVS and in identifying pairs of changed entities. Entities are usually files and the change is determined through observing additions or deletions of lines of code. Also, changes are interpreted between pairs of entities.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWPSE'07 September 3-4, 2007, Dubrovnik, Croatia
Copyright 2007 ACM 978-1-59593-722-3/07/09 ...$5.00.

In this paper, we propose a different approach, and we focus on identifying patterns of change that affect several entities in the same time. For this we use formal concept analysis [8]. Formal concept analysis is a technique that identifies sets of elements with common properties based on a given matrix that specifies the elements on the rows, properties on columns and the value of a field (i, j) is marked as true if the element i has property j.

To identify how entities changed in the same way, we use historical measurements to detect changes between two versions. For each history we identity each version in which a certain change condition is met. To use formal concept analysis, we use histories as elements, and "changed in version j"represents the jth property of the element.

Furthermore, for building the matrix of changes, we make use of logical expressions which combine properties with thresholds and which run on two versions of the system to detect interesting entities. In this way, we can detect changes that take into account several properties.

Example. ShotgunSurgery appears when every time we have to change a class, we also have to change a number of other classes [5]. We would suspect a group of classes of such a bad smell, when they repeatedly keep their external behavior constant and change the implementation. We can detect this kind of change in a class in the versions in which the number of methods did not change, while the number of statements changed.

One problem of concept analysis lies in the large number of results raised. To improve the precision of the detection we also propose a novel heuristic to filter out unwanted results.

Our approach can be applied at any level of abstraction. In this paper we propose several detections for packages, classes and methods. We provide evidence from initial experiments and a discussion of the approach based on several case studies.

Structure of the Paper. In the next section we briefly define two generic historical measurements. We describe Formal Concept Analysis in a nutshell in Section 3. We show how we use FCA to detect co-change patterns, and we show how we apply it on different levels of abstractions (Section 4). We discuss the results we obtained when applying our approach on a large open source case study (Section 6). In Section 9 we conclude and present the future work.

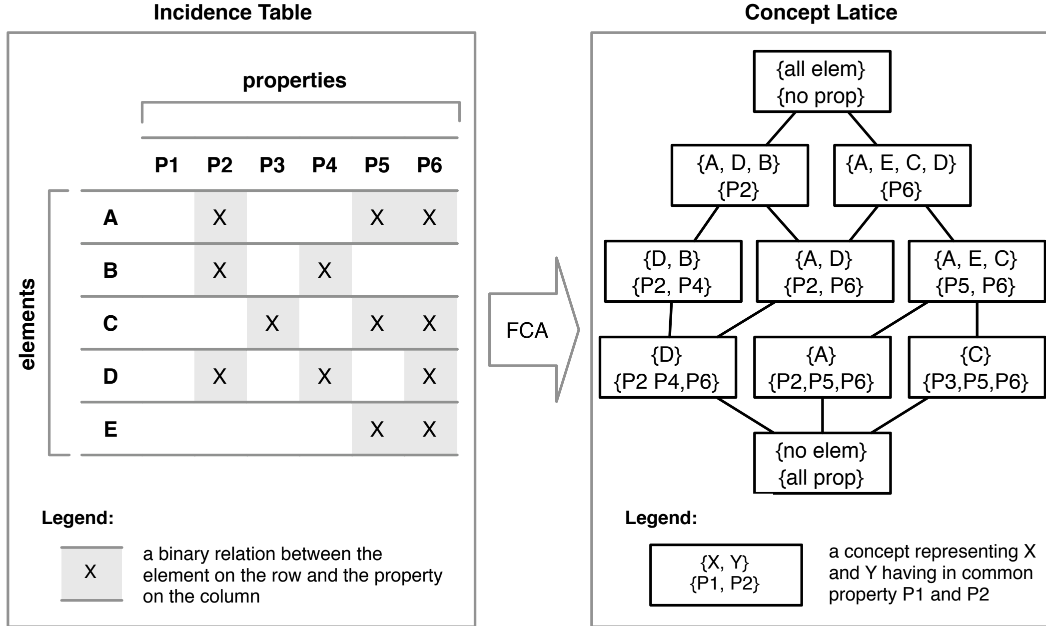

Figure 1: Example of applying formal concept analysis: the concepts on the right are obtained based on the incidence table on the left.

2. HISTORY MEASUREMENTS

Our approach is built on top of the Hismo meta-model [9, 10]. Hismo explicitly models history as a sequence of versions. We can have different types of histories based on the types of entities present in the structural meta-model (*e.g.,* PackageHistory, ClassHistory or MethodHistory).

To characterize how a version changed with respect to the previous one within a history, we can define several generic measurements. In this paper we use two generic historical measurements to distinguish between different types of changes.

Addition of a Version Property (A). We define a generic measurement, called addition of a version property P, as the addition of that property between version $i-1$ and i of the history H:

$$(i > 1) \quad A_i^H(P) = \begin{cases} P_i^H - P_{i-1}^H, & P_i^H - P_{i-1}^H > 0 \\ 0, & P_i^H - P_{i-1}^H \leqslant 0 \end{cases} \qquad (1)$$

Evolution of a Version Property (E). We define a generic measurement, called evolution of a version property P, as being the absolute difference of that property between version $i-1$ and i:

$$(i > 1) \quad E_i^H(P) = |P_i^H - P_{i-1}^H| \qquad (2)$$

We instantiate the above mentioned measurements by applying them on different version properties of different types of entities. For example, to identify if methods are added to a class, we will use NOM (number of methods) as P.

In this paper we make use of the several measurements:

- Method: *NOS* (number of statements), *CYCLO* (McCabe cyclomatic number [16]).
- Class: *NOM* (number of methods), *WNOC* (number of all subclasses).
- Package: *NOCls* (number of classes), *NOM* (number of methods).

The E measurement shows a change of a certain property, while the A measurement shows the additions of a certain version property.

3. CONCEPT ANALYSIS IN A NUTSHELL

Formal concept analysis is a technique that identifies meaningful groupings of elements that have common properties [8]. As an input, the technique requires a context that specifies a set of elements, a set of properties and a set of binary relationships between them.

Figure 1 gives a schematic example of the technique. The input is specified in the form of a so called incidence table which encodes binary relations between the set of elements and the set of properties. In our example, element A has three properties: P2, P5 and P6.

The output is a lattice of concepts, where each concept is a tuple of a set of elements and a set of common properties. The top concept in the lattice is the concept with all elements and no property. The more we move downwards the lattice the less elements and more properties we have, until we get to the bottom concept which contains no element and all properties. For example, elements A and D have two properties in common: P2 and P6.

Formal concept analysis is a generic technique working with elements and properties in general. To apply it in a particular context we need to map our interests on the elements and properties.

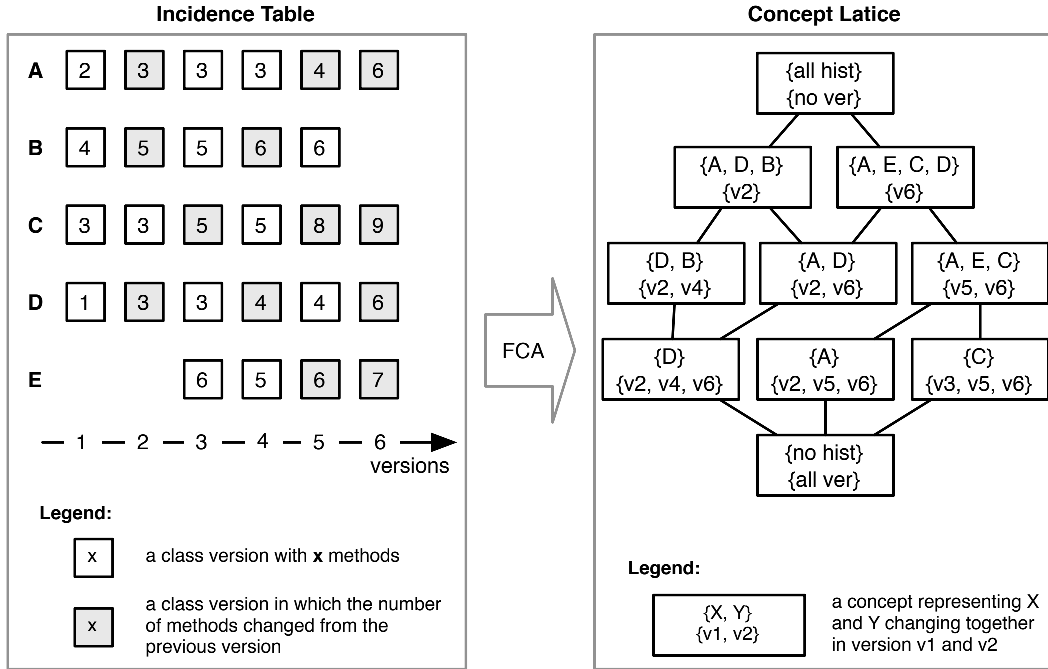

Figure 2: Example of applying concept analysis to group class histories based on the changes in number of methods. The Evolution Matrix on the left forms the incidence table where the property $\mathbf{P}_i$ of element X is given by "history X changed in version i."

4. USING CONCEPT ANALYSIS TO IDENTIFY CO-CHANGE PATTERNS

We apply formal concept analysis to detect co-change patterns. For this, we need to define elements, properties and the binary relationships between them: As elements we consider histories, and as a property we consider the predicate "changed in version j". Thus, a history has property j if the corresponding history has changed in version j.

We depict in Figure 2 an exemplification of the approach. To the left, instead of a table, we use the notation of an Evolution Matrix [13] in which each square represents a class version and the number inside a square represents the number of methods in that particular class version. A grayed square shows a change in the number of methods of a class version as compared with the previous version ($E_i(NOM) > 0$).

We use the matrix as an incidence table, where the histories are the elements and the properties are given by "changed in version j". Based on such a matrix we can build a concept lattice. To the right side of figure we show the concept lattice obtained from the Evolution Matrix on the left.

Each concept in the lattice represents all the class histories which changed certain properties together in those particular versions. In the given example, class history A and D changed their number of methods in version 2 and version 6.

To identify a change we want to be able to take into account several properties, and not only one. For example, to detect parallel inheritances it is enough to just look at the number of children of classes; but, when we want to look for classes which need to change the internals of the methods in the same time without adding any new functionality, we need to look for classes which change their size, but not the number of methods.

We encode this change detection in expressions consisting of logical combination of historical measurements. These expressions are applied at every version. In the example from Figure 2, we used as expression $E_i(NOM) > 0$ and we applied it on class histories.

4.1 Improving the detection precision

The lattice obtained from applying concept analysis can reveal many concepts, and from these only a small fraction are useful patterns, thus just considering all concepts reveals a low precision.

One reason for having many concepts is due to entities that have too many properties. One rule of thumb when applying concept analysis is to filter out from the table the elements that have all properties, given that they will appear in all concepts. However, when an element has almost all possible properties it will still appear in most of the concepts, thus not being necessarily relevant for a particular pattern.

We propose a novel heuristic to deal with this problem. Our goal is to detect patterns that document how a change to one entity should imply changes to other entities. Thus, we eliminate from a concept the entities that change in much more versions than in the concept (where "much more" is specified by a threshold value), because there is not necessarily a cause and effect link between this particular entity and the rest of the entities in the concept:

$$FilteringRule : \frac{conceptVersions}{totalChangedVersions} > threshold \quad (3)$$

Example. Suppose we have four entities A,B,C,D, where A and B changed in P2,P3,P4,P5,P6 and C and D changed in P5 and P6. In this case, we will have a concept ({A,B,C,D} {P5,P6}). However, A and B change in much more versions than C and D, and hence their change is not necessarily tied to the changed in C and D. Thus, we remove them from the concept.

After stripping the concepts of irrelevant entities, we remove all concepts that contain less than two entities, because "co-change" implies at least two entities.

5. CO-CHANGE PATTERNS

In the following sections we propose several expressions to detect co-change patterns at different levels of abstractions: methods, classes and packages.

5.1 Method Histories Patterns

Parallel Complexity. A set of methods are effected by *Parallel Complexity* when a change in the complexity in one method involves changes in the complexity of other methods. As a measure of complexity we used the McCabe cyclomatic number which indicates the number of different paths through a method. Classes with parallel complexity could reveal parallel conditionals.

$$ParallelComplexity : (A_i^H(CYCLO) > 0) \quad (4)$$

Dispersed logic. When methods change, but they do not add in complexity, the change is merely a modification of the current execution flow. This type of change can also be a hint of a bug fix.

Furthermore, when more such changes are spread over several methods, we can relate those methods as being part of the same logic. We name such a pattern *Dispersed logic*, and its presence might give indications of similar implementation which could be factored out. As an implementation measure we used number of statements:

$$DispersedLogic : (E_i^H(NOS) > 0) \wedge E_i^H(CYCLO) = 0) \quad (5)$$

5.2 Class Histories Patterns

Shotgun Surgery. The *Shotgun Surgery* bad-smell is encountered when a change operated in a class involves a lot of small changes to a lot of different classes [5]. To detect such a bad smell, we identify classes which do not change their interface (their number of methods remain constant), but change their implementation (their number of statements changes).

$$ShotgunSurgery : (E_i^H(NOM) = 0 \wedge E_i^H(NOS) > 0) \quad (6)$$

Parallel Inheritance. *Parallel Inheritance* is detected in the classes which change their number of children together [5]. Such a characteristic is not necessary a bad smell, but gives indications of a hidden link between two hierarchies.

For example, if we detect a main hierarchy and a test hierarchy as being parallel, it gives us indication that the tests were developed in parallel with the code.

$$ParallelInheritance : (A_i^H(WNOC) > 0) \quad (7)$$

Parallel Semantics. Methods specify the semantics of a class. With *Parallel Semantics* we detect classes which add methods in parallel. Such a characteristic could reveal hidden dependencies between classes.

For example, in test driven development, for each use-case we can have a test method. In these cases, we will detect test classes and system classes as being linked through parallel semantics.

$$ParallelSemantics : (A_i^H(NOM) > 0) \quad (8)$$

5.3 Package Histories Patterns

Package Parallel Semantics. If a group of classes is detected, as having parallel semantics, we would want to relate the containing packages as well. *Package Parallel Semantics* detects packages in which some methods have been added, but no classes have been added or removed.

$$PackageParallelSemantics : (E_i^H(NOCls) = 0) \wedge (A_i^H(NOM) > 0) \quad (9)$$

6. EXPERIMENTS

We have implemented our approach based on the Hismo meta-model [10] and as part of the Moose infrastructure [17]. We performed initial experiments using our approach. We applied it to two case studies and we report here some of the findings (see Table 1).

6.1 Parallel Inheritance in JBoss

The first case study consists of 41 versions of JBoss[1]. JBoss is an open source J2EE application server written in Java. The versions we selected for the experiments are at two weeks distance from one another starting from the beginning of 2001 until the end of 2002. The first version has 632 classes, the last one has 4276 classes (we took into consideration all test classes, interfaces and inner classes).

[1] See `http://www.jboss.org`.

System	Language	Versions	First Version	Last Version
JBoss	Java	41	40 kLOC 632 classes	281 kLOC 4276 classes
ArgoUML	Java	18	75 kLOC 1047 classes	95 kLOC 1587 classes

Table 1: Characteristics of the case studies.

Due to limited information in the parsed models, we were only able to perform experiments on parallel inheritance on this case study.

Applying the parallel inheritance detection without any filtering, reveals 68 concepts of class histories which added subclasses in the same time. Manual inspection showed there were a lot of repetitions (due to the way the concept lattice is built), and just a limited number of groups were useful.

For example, in 19 versions a class was added in the JBossTestCase hierarchy (JBossTestCase is the root of the JBoss test cases). Another example is ServiceMBeanSupport which is the root of the largest hierarchy of JBoss. In this hierarchy, classes were added in 18 versions. That means that both JBossTestCase and ServiceMBeanSupport were present in a large number of concepts, but they were not necessarily related to the other classes in these concepts.

These results showed that applying only concept analysis produced too many false positives. That is why we added a filtering step as described in Section 4.1.

For example, if JBossTestCase was part of a group of classes which changed their number of subclasses in 10 versions, we would rule the class out of the group. We chose an aggressive threshold (0.75) to reduce the number of false positives as much as possible, in the detriment of having true negatives.

After the filtering step, we obtained just two groups. In Table 2 we show the class histories and the versions in which they changed the number of children.

Class histories	Versions
org::jboss::system::ServiceMBeanSupport org::jboss::test::JBossTestCase	24 27 28 29 30 32 33 34 37 38 39 40 41 19 20
javax::ejb::EJBLocalHome javax::ejb::EJBLocalObject	24 41 28 30 32 36 37 38 23

Table 2: Parallel Inheritance in JBoss

In the first group we have two classes which changed their number of children 15 times: ServiceMBeanSupport and JBossTestCase. The interpretation of this group is that the largest hierarchy in JBoss is highly tested.

The second group detects a relationship between the EJB interfaces: EJBLocalHome and EJBLocalObject. This is due to the architecture of EJB which requires that a bean has to have a Home and an Object component.

6.2 Patterns in ArgoUML

The second case study consists of 18 versions of ArgoUML, a UML case tool written in Java. We selected versions 3 months distant, starting with the beginning of 2003 until May 2007.

We have applied all the proposed detections on the ArgoUML case study. We list some of the findings: in Table 3 we show the class histories that are linked through Parallel Inheritance, in Table 4 we show Parallel Semantics in different class histories, and in Table 5 we show the Method Histories that have increased their complexity in the same time. At this point we have performed no manual validation of the detected patterns.

Class histories	Vers.
application.events.ArgoNotationEventListener kernel.DelayedVChangeListener org::argouml::ui::TabTarget	5 8 10 18 14 3 15 7 9 12
application.events.ArgoNotationEventListener ui.targetmanager.TargetListener kernel.DelayedVChangeListener	5 8 10 18 14 15 7 9 11 12 16

Table 3: Parallel Inheritance in ArgoUML

Class histories	Vers.
uml.diagram.static_structure.ui.UMLClassDiagram uml.diagram.ui.UMLDiagram	2 18 14 12 8 3 11 9
uml.diagram.deployment.ui.FigMNode uml.diagram.deployment.ui.FigComponent	14 12 9
uml.diagram.state.ui.FigState uml.diagram.state.ui.FigTransition	13 12 8 3
uml.diagram.ui.FigEdgeModelElement uml.diagram.ui.FigNodeModelElement	10 17 5 7 4 15 2 14 12 8 3
persistence.ZargoFilePersister persistence.ModelMemberFilePersister	16 10 17
uml.ui.foundation.core.PropPanelClassifier uml.ui.UMLMutableLinkedList	2 8 3
uml.diagram.deployment.ui.FigMNodeInstance uml.diagram.deployment.ui.FigObject	13 14 12
uml.diagram.collaboration.ui.UMLCollaborationDiagram uml.diagram.deployment.ui.UMLDeploymentDiagram uml.diagram.use_case.ui.UMLUseCaseDiagram	2 18 12 9
uml.diagram.ui.FigAttributesCompartment uml.diagram.ui.FigOperationsCompartment	13 15 11

Table 4: Parallel Semantics in ArgoUML

7. DISCUSSION

On modeling history as first class entity. Having history as a first class entity, allowed a straight forward mapping to the elements of the incidence table. To identify the

Method histories	Vers.
FigComponentInstance.setEnclosingFig(Fig) FigComponent.setEnclosingFig(Fig) FigMNodeInstance.setEnclosingFig(Fig)	8 10 17
ModelFacade.getName(Object) FigNodeModelElement.setEnclosingFig(Fig)	4 5 7 9

Table 5: Parallel Complexity in ArgoUML

xth property, we computed for the *xth* version the expression detecting the change. Having historical properties made it easy to encode the expressions. Below we give the OCL code expressed on Hismo for the ShotgunSurgery expression defined for a ClassVersion:

```
context ClassVersion

  -- returns true if the the number of methods did not change
  -- and the number of statements changed
  -- with respect to the previous version
  derive hasShotgunSurgerySymptom:
     (self.ENOM = 0) &
     (self.ENOS > 0)
```

On versions sampling. For detecting co-change patterns the changes need to be identified within the same version. However, in the case some of the patterns, like Parallel Inheritance, it is not necessary to have the changes exactly in the same version. Thus, the granularity of versions plays an important role in the detection. One possible solution is, given a sum of available versions, to automatically perform the detection by with different versions samplings (*e.g.,* every second or every third version).

On the filtering algorithm. To improve the precision of concept analysis, we employ a filtering algorithm to remove false positives. According to our algorithm the effectiveness of the approach is highly affected by the value of the threshold. When the threshold is high (*i.e.,* close to 1) we aggressively remove the false positives but we risk missing true negatives. Further work is required to identify the best value for the threshold.

8. RELATED WORK

The first work to study the entities that change in the same time was performed by Gall *et al.* [6]. The authors used the co-change information to define a proximity measurement which they use to cluster related files. The work has been followed up by the same authors [7] and by Itko *et al.* [12]. Bouktif *et al.* improved the co-change detection by using dynamic-time warping [1]. Shirabad *et al.* looked at the same information and employed machine learning techniques to detect files which are likely to need to be changed when a particular file is changed [18].

These approaches place the analysis at the file level. As opposed to these previous approaches, Zimmerman *et al.* placed their analysis at the level of classes and methods [21, 22]. Their focus was to provide a mechanism to warn developers that: "Programmers who changed these functions also changed ...". Their approach differs from ours because they only look at syntactic changes, while we identify changes based on the semantics of the changes. Furthermore, our approach takes into consideration several changes in the same time.

Breu and Zimmermann also took the semantics of changes into account when they devised an approach to identify aspects [2]. In particular they looked at additions of the same method calls in multiple places and in the same versions to identify candidates for aspects. Their goal is different than ours, as we aim at identifying patterns by detecting repetitive changes.

Part of the patterns we detect are signs of design flaws (*e.g.,* Shotgun Surgery). Detection of problems in the source code structure has long been a main issue in the quality assurance community. Marinescu [15] detected design flaws by defining detection strategies. Ciupke employed queries usually implemented in Prolog to detect "critical design fragments" [3]. Tourwe *et al.* also explored the use of logic programming to detect design flaws [19].

van Emden and Moonen detected bad smells by looking at code patterns [20]. These approaches differ from ours because they use only the last version of the code, while we take into account historical information. Furthermore, van Emden and Moonen proposed as future research the usage of historical information to detect Shotgun Surgery or Parallel Inheritance. In a position paper, Davey and Burd proposed the usage of concept analysis to detect such evolutionary concepts [4].

9. CONCLUSIONS

Understanding how a system changes can reveal hidden dependencies between different parts of the system. Moreover, such dependencies might reveal bad smells in the design.

Analyzing the history of software systems can reveal parts of the system that change in the same time and in the same way. We proposed the usage of formal concept analysis, a technique that identifies elements with common properties based on an incidence table specifying binary relations between elements and properties.

To detect the changes in a version, we used expressions that combine different properties to detect complex changes. By applying these queries on every version we obtained an Evolution Matrix annotated with the change information which we then used as input for a concept analysis machine. In other words, we used as elements histories and as properties we used the knowledge of "changed in version i". The results were groups of histories that change together and the versions in which they changed.

An important contribution of our approach is given by the heuristic to automatically filter the raw results of the concept analysis machine: a history is relevant to a concept, if it was not changed in many more versions than the ones in the concept.

We have started to use our approach on several case studies and we reported here some of the results of the initial experiments. In the future we plan to apply our approach on more case studies and analyze in depth the results we obtain at different levels of abstraction. One particular focus of the future experiments will be the impact of the threshold of the filtering algorithm on precision and recall.

Acknowledgments. Gîrba, Kuhn and Marinescu gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "NOREX — Network of Reengineering Expertise" (SNF Project IB7320-110997).

10. REFERENCES

[1] S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol. Extracting change-patterns from cvs repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 221–230, 2006.

[2] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[3] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.

[4] J. Davey and E. Burd. Clustering and concept analysis for software evolution. In *Proceedings of the 4th international Workshop on Principles of Software Evolution (IWPSE 2001)*, pages 146–149, Vienna, Austria, 2001.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[6] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[7] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

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

[9] T. Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, Nov. 2005.

[10] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[11] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[12] J. Itkonen, M. Hillebrand, and V. Lappalainen. Application of relation analysis to a small Java software. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 233–239, 2004.

[13] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Langages et Modèles à Objets (LMO'02)*, pages 135–149, Paris, 2002. Lavoisier.

[14] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[15] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timişoara, 2002.

[16] T. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[17] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[18] J. S. Shirabad, T. C. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *International Conference on Software Maintenance (ICSM 2003)*, pages 95–104, 2003.

[19] Tom Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003)*, pages 91–100. IEEE Computer Society Press, Mar. 2003.

[20] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, Oct. 2002.

[21] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.

[22] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.