Refining the Architecture Recovery Approach ArchMine by Incrementally Performing Evaluation Studies

Aline Vasconcelos[1, 2] and Cláudia Werner[1]

[1] Federal University of Rio de Janeiro
COPPE/UFRJ – Systems Engineering and Computer Science Program
P.O. Box 68511 – ZIP 21945-970 – Rio de Janeiro – RJ – Brazil

[2] CEFET Campos - Federal Center for Technological Education of Campos
Dr. Siqueira, 273 – ZIP 28030-130 – Campos dos Goytacazes – RJ- Brazil

apires@cefetcampos.br, werner@cos.ufrj.br

Abstract. *Several architecture recovery approaches have been proposed in the literature with distinct goals. ArchMine, an architecture recovery approach based on dynamic analysis and data mining, aims to assist in program understanding and software reuse by detecting cohesive classes that implement a set of related functionalities, i.e. architectural elements. It is supported by a tool set integrated into a reuse based software development environment. The approach and its tool set have been detailed in previous works. The focus of this paper is to describe how ArchMine and its tool set were evaluated and refined through a series of evaluation studies. These evaluations allowed to gradually refine the approach based on the lessons learned.*

1. Introduction

There is a large number of reverse engineering approaches in the literature to recover documentation from existing systems. Approaches of software clustering and remodularization, e.g. (MITCHELL & MANCORIDIS, 2006), or more specifically of architecture recovery, e.g. (KAZMAN & CARRIÈRE, 1997; ANQUETIL & LETHBRIDGE, 1999; BOJIC & VELASEVIC, 2000; SARTIPI, 2003), lead to the reconstruction of documentation from system available artifacts, such as source code and executables. They are motivated by the fact that existing systems, which have been developed many years ago (i.e. legacy systems), usually do not have an up-to-date documentation that can help in their understanding and reuse. These systems usually represent a great investment made by the organizations and incorporate business knowledge that sometimes cannot be obtained in any other source of information. However, in general, existent reverse engineering approaches are not focused on reuse and are based on criteria that are domain or implementation specific.

In the last years, architecture recovery has been receiving a lot of attention from the reverse engineering community, since architectures communicate high-level knowledge about systems, facilitating their comprehension and reuse. Architecture recovery can be defined as a reverse engineering activity that aims at obtaining a documented architecture for an existing system (DEURSEN *et al.*, 2004). Many definitions to software architecture are found in the literature since it has not consolidated concepts and

representations yet. Among the accepted definitions, we adopt the one from Bass *et al.* (BASS *et al.*, 2003), where software architecture is defined as the structure, or structures of a system that comprise software elements (i.e. architectural elements), the externally visible properties of those elements, and the relationships among them.

The recovered architectural element in this work stands for a group of functionally cohesive classes that implement a domain concept and that can later be used for reuse purposes. Concerning architectural structures or views, our approach, i.e. ArchMine (VASCONCELOS, 2007; VASCONCELOS & WERNER, 2007a), recovers a static and a dynamic architectural representation. The architecture is represented in UML, where the static view is represented by a package diagram, each package representing an architectural element with its internal structure composed by a set of related classes. The dynamic view is represented through sequence diagrams extracted at the architectural level, showing messages exchange among architectural elements.

ArchMine is supported by the TraceMining tool, a plugin of the Odyssey environment (ODYSSEY, 2007), which is a reuse infrastructure based on domain models that supports both: development for reuse, through a Domain Engineering (DE) process; and development with reuse, through an Application Engineering (AE) process. ArchMine is integrated to the AE context and aims at generating artifacts that can later be reused in a DE process. In order to be reusable, it is necessary to ensure that the recovered architectural elements are cohesive and consistently represent domain concepts. To this end, ArchMine adopts an architectural evaluation method based on inspection, i.e. ArqCheck (BARCELOS & TRAVASSOS, 2006), extended in our work to evaluate the reusability of architectural elements (VASCONCELOS & WERNER, 2007b).

Our goals in this work are: (1) to contribute to software reuse by clustering functionally cohesive classes that represent domain concepts and possibly reusable artifacts; (2) to provide an architecture recovery approach that can be reused across different domains and implementations; (3) to incorporate an architectural evaluation method to the reverse engineering approach, assuring the quality of the recovered architecture with regard to the established goals. These goals were evaluated through 4 evaluation studies that allowed us to gradually refine ArchMine and its supporting tool set.

ArchMine is based on use case scenarios, dynamic analysis, and data mining. The first represent usage scenarios to guide application execution during dynamic analysis. It results in a set of execution traces (i.e. method executions) related to application functional requirements and represent a means to map them to system entities (i.e. classes). Data mining is employed to mine the gathered execution traces and discover related classes based on the functionalities they implement. ArchMine is semi-automated and human-guided. We argue that in contrast to fully automated approaches (e.g. Bunch (MITCHELL & MANCORIDIS, 2006)), by incorporating human knowledge, we can extract models that are richer of information and directed to user goals, e.g. software reuse. ArchMine has been detailed in previous works and therefore the focus of this paper is to present the performed evaluation studies that allowed to gradually refine it and its corresponding tool set.

This paper is organized as follows: Section 2 presents meaningful related work in architecture recovery and architecture evaluation; Section 3 describes our approach for architecture recovery; Section 4 describes the studies conducted to evaluate the ap-

proach and its tool set, detailing the last evaluation study performed within an industrial context; and Section 5 concludes the paper presenting the contributions and limitations of the approach, and some future work.

2. Related Work

Concerning architecture recovery, works on software clustering and remodularization, besides architecture recovery itself, have many points in common. In (ANQUETIL & LETHBRIDGE, 1999), an architecture recovery approach based on clustering classes by name similarity is presented. According to the results of experimental studies they could get evidences that name similarity can be a good criterion based on static analysis to architecture recovery. Bojic and Velasevic (2000) apply dynamic analysis to architecture recovery as the work presented in this paper, but using a formal concept analysis technique. However, there are no evaluation studies to confirm the validity of their approach, as we do in our work. In (SCHMERL *et al.*, 2006), dynamic analysis is also employed, but in this case to map low-level system events into more abstract architectural operations or connectors. Mappings are formally defined using Colored Petri Nets. Although proposing a language to define the mappings and relying on regularities in system implementations, these mappings are very implementation dependant. Our approach, on the other hand, is not based on implementation patterns.

Kazman and Carrière (KAZMAN & CARRIÈRE, 1997) present a semi-automated approach to architecture recovery performed with the support of the Dali workbench. In their approach, criteria to classes clustering into architectural elements are application-dependant, while in our approach they are general to object oriented (OO) applications from different domains. Dali has evolved to the Armin (O'BRIEN & STOERMER, 2003) environment, which provides more powerful architectural views manipulation, although not evolving clustering criteria. Sartipi (SARTIPI, 2003) also employs data mining to architecture recovery, but in his work data mining is used to derive relations among entities based on their static structural properties, instead of their dynamic behavior as in our approach.

Mitchell and Mancoridis (MITCHELL & MANCORIDIS, 2006) present an automatic approach to architecture recovery that analyzes the static graph extracted from source code and clusters its entities based on the evaluation of their coupling degree. They apply a sub-optimal function that tries to maximize the connectivity inside architectural elements, while minimizing the connectivity among architectural elements. Architectural element names are derived randomly by selecting the name of one of their constituent classes. This problem of not automatically attributing semantic names to architectural elements is general in reverse engineering approaches, with the exception of some works such as the one presented in (TZERPOS, 2001).

To the best of our knowledge, current reverse engineering approaches do not provide a systematic means to evaluate the recovered architectures, being informally evaluated by system experts. In order to fulfill this gap, we adopt an extended version of ArqCheck (BARCELOS & TRAVASSOS, 2006) to evaluate the architectures recovered by ArchMine. ArqCheck is a generic and simple approach to architecture evaluation as described in Section 3.2. Some architecture evaluation methods, as, for example, the ones based on scenarios - e.g. ATAM (KAZMAN *et al.*, 2000) and SAAM (KAZMAN

et al., 1994), may require a great effort, and others are specific to some architectural representations – e.g. SAEM (DUENAS *et al.*, 1998).

3. The Architecture Recovery and Evaluation Approach

In this Section we present our approach for architecture recovery and evaluation. First, architecture recovery takes place, encompassing information extraction and architectural views reconstruction. Architecture evaluation main goal is to evaluate the semantics, consistency, and reusability of the reconstructed architectural elements. Based on its results, recovered architectural elements can be restructured. These processes jointly derive knowledge about the application to help in its understanding and reuse.

3.1. ArchMine: Architecture Recovery based on Mining Execution Traces

In order to reconstruct architectural elements, classes are clustered based on relations among them, derived by means of dynamic analysis and data mining. ArchMine is application and implementation independent, which means that it can be applied to OO applications from different domains with distinct implementation patterns. However, its supporting tool set was designed to analyze Java applications. The remainder of this Section explains ArchMine activities and its tool set. Further details and examples can be found in (VASCONCELOS, 2007; VASCONCELOS & WERNER, 2007a).

3.1.1. Static Structure Extraction and Use Case Scenarios Definition

ArchMine process is detailed in Figure 1. The process is modeled with OMG SPEM (Software Process Engineering Metamodel) notation. Its first 2 activities, i.e. *Static Structure Extraction* and *Use Case Scenarios Definition*, can be performed in parallel.

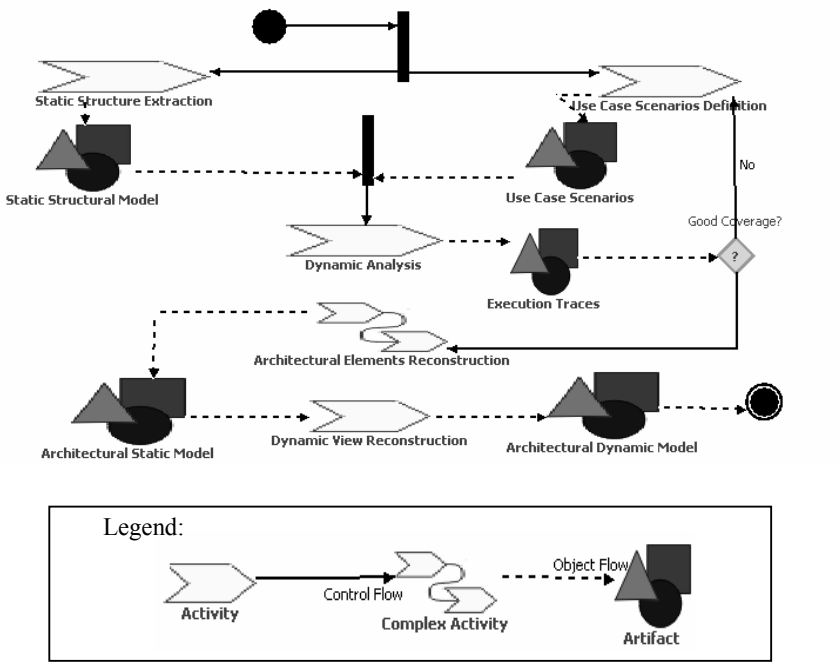

Figure 1. Architecture recovery process

Static Structure Extraction reconstructs a UML package and class model from source code. This task is fully automated by applying the Odyssey static reverse engineering tool, Ares. This recovered class model is at a low-abstraction level and its classes are further restructured into architectural elements. *Use Case Scenarios Definition* aims at determining usage scenarios to guide application execution in the dynamic analysis. A use case represents a functional requirement of a system and each sequence of actions in a use case is called a use case scenario, defining one way for obtaining that functional requirement. In ArchMine, some guidelines are defined to help in the definition of use case scenarios, which were derived and refined through the evaluation studies in which the approach was applied. They must be specified in the Odyssey environment. ArchMine does not require 100% of coverage of scenarios definition. However, the specified use case scenarios have a great impact in the recovered architecture.

3.1.2. Dynamic Analysis

The next activity is *Dynamic Analysis*. It involves application execution and monitoring for the specified use case scenarios in order to gather execution traces related to the realization of an application functionality or use case scenario. We instrument the bytecode of Java applications through AspectJ, allowing the detection of methods execution (i.e. execution traces) with related information, i.e. class, instance and thread. It is performed with the support of our Tracer tool (VASCONCELOS, 2007), which takes as input the jar file and classPath of Java applications, and inserts the tracing code around each method execution. The output is a set of XML trace files (Figure 3), in which methods are ordered and indented by control flow hierarchy. Through static filters, Tracer eliminates from the execution traces classes belonging to libraries (e.g. Java API), which are not relevant to comprehend the application architecture. Since Tracer allows enabling and disabling tracing, it is possible to delimit a set of method executions, i.e. execution trace, belonging to a use case scenario.

In order to evaluate classes coverage, the classes in the execution traces are compared to the ones extracted in the static model. Classes that were not monitored are shown to a system expert (e.g. a programmer, a developer, a designer) who indicates scenarios that still need to be executed. As shown in Figure 1 there is an iteration between the *Dynamic Analysis* and *Use Case Scenarios Definition* activities.

3.1.3. Architectural Elements Reconstruction

Architectural Elements Reconstruction is a complex activity (Figure 1), composed of a series of tasks, namely: classes clustering, architectural elements names generation, and architectural elements dependencies computation. We apply a data mining algorithm adapted from Apriori (AGRAWAL & SRIKANT, 1994) to mine execution traces and detect high-level relations among classes that implement common functionalities in the system. Apriori was chosen because it allows discovering recurrent relations among data items. Its main concepts are:

- Apriori discovers association rules among items of transactions. Association rule is an implication of the form: $X \Rightarrow Y$, where X and Y are items of the database and $X \cap Y = \varnothing$. X is the antecedent of the rule, while Y is its consequent. Our rules have one antecedent and many consequents.

- Apriori requires 2 threshold values: minimum support and minimum confidence. Support "s" means that s% of the transactions of the database contain X and Y. Confidence "c" implies that c% of the transactions that contain X also contain Y. Given a set of transactions τ, the problem of mining association rules is to generate all rules that have support and confidence equals or greater than the user specified minimum percentages.

Table 1. Mapping of concepts for mining association rules

Data Mining Concepts	Mapping to Dynamic Analysis
Transaction	A use case scenario, represented by an execution trace.
Data Item	A class, in the execution trace, implementing the use case scenario.
Support	Percentage of use case scenarios implemented by a class.
Minimum Support	The minimum percentage of use case scenarios in which the classes in an association rule must appear together.
Confidence	Percentage of use case scenarios of class X in which a class Y also appears.
Minimum Confidence	The minimum percentage of use case scenarios of class X in which a class Y must also appear for the association rule between X and Y to be valid.
Antecedent	The class that is used as input to discover the association rules.
Consequent	The classes that are associated to the antecedent with support and confidence greater or equals to the minimum values.

In order to mine association rules, some concepts from the database domain were mapped to the dynamic analysis context. These mappings are presented in Table 1. Mining is supported by our TraceMining tool – Figure 2, and is performed in many mining cycles. TraceMining reads the gathered execution traces and generates architectural elements in the Odyssey environment. Instead of detecting large itemsets in the database and then deriving association rules among them, as in the original Apriori, TraceMining queries specific antecedents, which are randomly chosen by the tool. Therefore, association rules are discovered for these items.

Mining is guided by an heuristic set supported by the TraceMining tool. These heuristics were incrementally derived based on the lessons learned of the evaluation studies in which ArchMine was applied (see Section 4) and are detailed in the following: **(H1)** minimum support must be low, since the most monitored classes clustered into architectural elements the higher tends to be the quality of the recovered architecture – our algorithm is mostly based on relative probability of classes occurrence (i.e. confidence) instead of an absolute probability (i.e. support); **(H2)** minimum confidence must be tuned along the mining process by the developer who recovers the architecture (e.g. 60% in Figure 2); **(H3)** classes must be mined from higher to lower support values, allowing the detection of more general architectural elements that provide services required by many others (e.g. classes support from 70 to 100% for antecedents selection in Figure 2); **(H4)** classes that participate in few use case scenarios tend to compose more specific business architectural elements; **(H5)** classes that participate in already formed groups must be filtered from subsequent mining cycles (see the checkbox at the bottom of Figure 2); **(H6)** superclasses and interfaces may not be monitored during dynamic analysis since they do not receive direct messages at run time, and, therefore, they must be statically grouped in the architectural element to which they have the highest coupling (i.e. superclasses will be grouped with the most of their subclasses and interfaces with the most of their realizing classes). This heuristic set does not intend to be an exhaustive list, although it proved to provide good results along the performed evaluation studies.

Reconstructed architectural elements are exported to Odyssey by the TraceMining tool – Figure 2, restructuring the already extracted static model and leading it to a higher abstraction level. As shown in Figure 2, TraceMining implements the described heuristics of ArchMine. For example, H3 and H4 stands for the support of classes used as antecedents for mining. H6, on the other hand, is implemented during the exportation process, since it is based on static properties and is not explicitly presented by the tool.

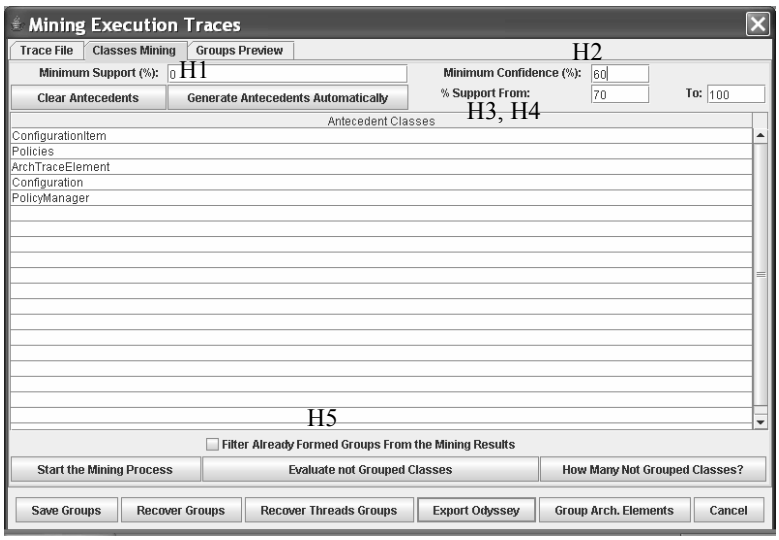

Figure 2. TraceMining mining input data

During exporting, architectural element names are derived. This is done by breaking down the names of their classes into substrings and ranking their occurrences. Class names are broken down into substrings that start with a capital letter or after an underscore symbol. TraceMining establishes a ranking among the substrings by counting their occurrences in architectural element class names, and composes architectural element names by concatenating the best ranked ones. In many cases, this approach achieves good results, since architectural elements tend to have classes with common substrings in their names, as observed in the performed evaluation studies. But, in other cases, it does not generate adequate semantic results. The user can ever change the generated names. Besides deriving element names, TraceMining also computes dependencies among architectural elements based on the relationships among their classes.

Tables 2 and 3 present a hypothetical school example of the heuristics usage. According to heuristics H3 and H4, classes are mined from higher to lower support values, starting from Student class (100% of support), that appears in all execution traces (Table 2), and finishing with PrinterUtils class (33,3% of support). Already grouped classes are filtered from subsequent mining cycles (H5). Therefore, Student, GraduateStudent, and UnderGraduateStudent are filtered from mining cycles 2 and 3. Minimum confidence used was 60% (H2), a value that was tuned along the evaluation studies (Section 4). Then GraduateStudent and UnderGraduateStudent were suggested for clustering with Student since they appear in 66,7% of the use case scenarios that Student appears. At the end, 3 architectural elements were recovered, namely: Student, ClassesSubscription, and Printer (Table 3). Grade class was not covered by any element and can be manually clustered in the architecture, since it has not a relative probability

of occurrence greater than or equals to 60% for any other class (H2).

In Figure 3, it is summarized the integration scheme of the tool set that supports ArchMine. Ares exports an extracted class model from legacy systems to Odyssey, where use cases and their scenarios are also specified. The Tracer tool takes as input use case scenarios and the jar file of Java applications to gather execution traces at run time and generates XML execution traces associated to use case scenarios. TraceMining tool reads the execution traces and restructures the previously extracted model in the Odyssey environment, based on mining results, leading it to the architectural level.

Table 2. Use case scenarios x execution traces classes for a school application

Use Case scenario	Classes in the Correspondent Execution Traces
1. Inform Graduate Students Grade	Student, GraduateStudent, Grade
2. Inform Undergraduate Students Grade	Student, UndergraduateStudent, Grade
3. Print Students Grade	Student, GraduateStudent, UndergraduateStudent, PrinterUtils, PrinterConfig, Grade
4. Subscribe Graduate Students	Student, GraduateStudent, Classes, Subscription
5. Subscribe Undergraduate Students	Student, UndergraduateStudent, Classes, Subscription
6. Print Students Subscriptions	Student, GraduateStudent, UndergraduateStudent, Classes, Subscription, PrinterUtils, PrinterConfig

Table 3. Mining cycles

Cycle	Antecedent	Support	Association Rules/Confidence
1	Student	100%	Student ⇒ GraduateStudent, UndergraduateStudent –66,7%
2	Classes	50%	Classes ⇒ Subscription – 100%
3	PrinterUtils	33,3%	PrinterUtils ⇒ PrinterConfig – 100%

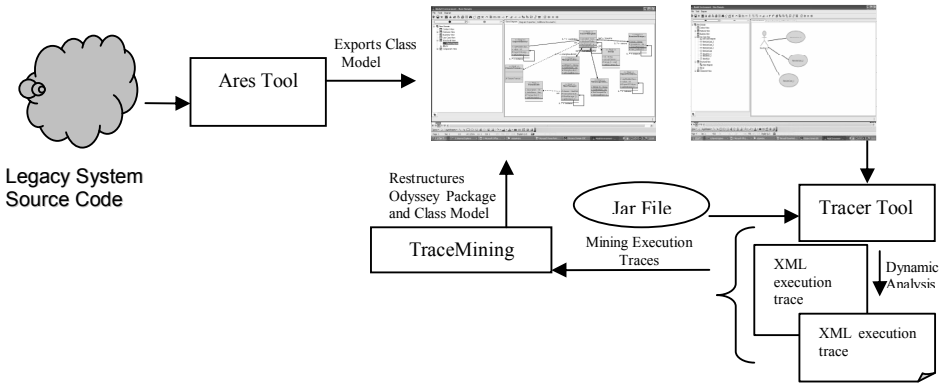

Figure 3. Tool set integration scheme that supports ArchMine approach

3.1.4. Dynamic View Reconstruction

Dynamic View Reconstruction aims at representing interactions among architectural elements, showing how the system behaves in its architecture to perform use case scenarios. Moreover, it allows establishing traceability links between use case scenarios (i.e. application functionalities) and architectural elements. The dynamic view is represented through UML sequence diagrams and its reconstruction is performed with the support of our Phoenix tool. This phase of the approach was not evaluated through evaluation studies yet. For this reason it will not be detailed in this paper. Further details about this activity of ArchMine can be obtained in (VASCONCELOS, 2007).

3.2. ArqCheck: Architecture Evaluation based on Inspection

The recovered architecture is evaluated by applying an extended version of ArqCheck

(VASCONCELOS & WERNER, 2007b). ArqCheck is a manual architectural inspection method that uses a checklist as the defect detection technique. It was chosen due to: it is simple and it requires less effort to architecture evaluation than other methods, such as the ones based on scenarios (KAZMAN *et al.*, 1994; KAZMAN *et al.*, 2000); it can be configured to the architectural representation at hand; its feasibility was evaluated through 2 evaluation studies previously performed; and it has been developed in the same academic environment as this work, facilitating its extension.

The questions are divided in 3 categories: architectural representation consistency, conformance to functional requirements, and conformance to non-functional requirements. For the non-functional requirements, the questions are based on knowledge available in architectural tactics (BASS et al., 2003), covering Availability, Performance, Modifiability, Usability, Security, and Testability. The checklist was extended in this work to evaluate architectural elements Reusability. Since there are no tactics for Reusability, the questions were formulated based on knowledge available in Component-Based Development literature (SAMETINGER, 1997), which aims at defining self-contained and cohesive software artifacts, with well-defined functionalities.

Moreover, in order to describe non-functional requirements, quality scenarios are adopted, representing an information set that allows characterizing a non-functional requirement that facilitates its understanding (BASS *et al.*, 2003). Quality scenarios describe non-functional requirements like use case scenarios describe the functional ones. Quality scenarios in the checklist must be instantiated any time it is applied, according to the non-functional needs of the current system. In an inspection process, 3 roles are involved, namely: the moderator, who manages the process execution; the inspected artifact author; and the inspector, who identifies defects on the artifact.

4. Evaluation Studies

ArchMine has been refined since 2005 through 3 evaluation studies, involving the recovery of the architectures of 5 applications with different sizes from various domains. Moreover, the extensions performed in ArqCheck were also evaluated. In the last study, performed within an industrial context, ArchMine was evaluated considering its integration with ArqCheck. Through the 4 conducted evaluation studies, it is possible to observe how ArchMine has evolved until the point it is described in this paper.

This Section is organized as follows: a historical view of the preliminary evaluation studies in this work is presented, stating the goals and some lessons learned in each of them; then, the last study, performed within an industrial context, is detailed with its goals, hypothesis, metrics, planning, threats to validity, execution, and results along with lessons learned. Further information can be found in (VASCONCELOS, 2007).

4.1. First Evaluation Study: ArchMine Feasibility

The goal of the first evaluation study was to characterize the feasibility of ArchMine in recovering comprehensible architectural models for object oriented applications. The evaluation study outline is presented as follows (WOHLIN et al., 2000): **Analyze** ArchMine architecture recovery approach; **For the purpose of** characterization; **With respect to** the feasibility of recovering a comprehensible architectural model for an OO application, where architectural elements represent domain concepts; **From the point of**

view of software engineers; **In the context of** the recovery of the architecture of 3 academic applications, for which there is an available system expert to evaluate.

The objects of the study were 3 applications developed in the same academic environment of this research. Although there was a bias in the study, it served to initially evaluate the approach with a minimal cost in order to get evidences about its feasibility. Performing the evaluation in an industrial environment would require a great effort and cost, thus it should be done when the approach is mature enough.

The applications had different sizes, complexities, and were from various domains. One of them was a game to train Software Managers, i.e. the TIM application (DANTAS *et al.*, 2006), with 51 classes. The second one was ArchTrace (MURTA *et al.*, 2006), a traceability links evolution system that maintains links between an architecture description and the related source code. It contains 80 classes. And the last one was a tool to the specification, verification, and validation of OCL constraints in UML models, Odyssey-PSW (CORREA & WERNER, 2004), with 596 classes. The goal of using applications with different characteristics was to minimize the bias in the study. With just one application it would not be possible to be sure if the results obtained were due to the application characteristics or due to the approach itself.

As the results of this study, there were evidences, according to the experts, that ArchMine could recover a reasonable architecture for OO applications. The experts evaluated each architectural element outlining how many classes were correctly allocated and how many were missing. Therefore, correctness (i.e. precision) and coverage (i.e. recall) were calculated. Comparing precision and recall with values of the literature (ANQUETIL *et al.*, 1999; SARTIPI, 2003), allowed us to get evidences of ArchMine feasibility, since these values were quite close. Moreover, the system experts filled evaluation forms, in which one of them agreed that ArchMine recovered a comprehensible architecture, and the other 2 found that it partially recovered the architecture.

Finally, some lessons learned, according to the experts evaluation, were: semantic names should be attributed to the architectural elements to facilitate system comprehension; a great effort was required to evaluate the recovered architectures; a great effort was also required to recover the architectures, i.e. 2 hours for the TIM games application, 4 hours for ArchTrace, and 30 hours for Odyssey-PSW, since the mining process was not completely supported by the TraceMining tool in this study; some superclasses and interfaces were not monitored and had to be manually allocated in the architecture.

4.2. Second Evaluation Study: a Comparative Study

In order to confirm ArchMine feasibility and to check whether the improvements made after the first evaluation study were worthwhile, a second study was performed. Some of these improvements were: an algorithm to derive names for the reconstructed architectural elements; TraceMining automatically suggesting antecedents for mining in a support boundary; a heuristic to allocate superclasses and interfaces in the architecture. In this study, ArchMine performance was compared to the one of another architecture recovery approach, i.e. Bunch (MITCHELL & MANCORIDIS, 2006). Bunch was chosen because it was the only approach for architecture recovery that had an available and scalable tool, besides being strongly referenced in the literature. Without an adequate support it is not possible to be sure if an approach is being correctly applied.

The outline of this study is: **Analyze** refined ArchMine and Bunch; **For the purpose of** characterization; **With respect to** the efficacy in recovering the architectures of object oriented applications; **From the point of view** of software engineers; **In the context of** the recovery of the architecture of 2 academic applications, for which there is an available system expert that can evaluate the recovered architectures.

In this study, the objects were ArchTrace, i.e. one of the applications used in the first study, and the Odyssey environment (ODYSSEY, 2007). ArchTrace has 80 classes and Odyssey 595 classes. The architectures of both applications were recovered by applying Bunch and ArchMine. It was not possible to compare the recovery effort, since Bunch is automated and ArchMine is not. However, comparing the performance of ArchMine against the one obtained in the first study, it was possible to observe that it had improved after the incorporated refinements. The recovery of the architecture of Odyssey took 10 hours against 30 hours for the recovery of the Odyssey-PSW architecture in the first study. These applications have almost the same size, i.e. around 590 classes. This performance improvement is due to the refinements of ArchMine and the higher automation of the TraceMining tool after the incorporation of the lessons learned in the first study. At this moment, TraceMining was already incorporating all the heuristics explained in Section 3.1.3. Moreover, the effectiveness of ArchMine also improved, since precision and recall values for ArchTrace, also used in the first study, were better. Although there is a bias concerning the same application used twice, ArchTrace architecture was recovered by strictly applying ArchMine heuristics.

At the end, it was observed that ArchMine got better results than Bunch from the point of view of the system experts concerning recovery efficacy. The precision for Odyssey architectural elements, for example, was 75% in ArchMine recovered architecture and 35% in Bunch recovered architecture. Coverage, on the other hand, was 48% with ArchMine against 24% of Bunch. This result takes into consideration that the goal of the recovery was to get architectural elements that should correctly represent domain concepts. However, architecture evaluation effort was still too high. The system expert of the Odyssey application took around 5 hours evaluating each of the recovered architectures. Therefore, the next studies goals were to evaluate if extended ArqCheck, the approach adopted for architecture evaluation in our work, could minimize this problem.

4.3. Third Evaluation Study: Extended ArqCheck Feasibility

The goal of this evaluation was to characterize the feasibility of extended ArqCheck to evaluate architectural elements Reusability. ArqCheck had been previously evaluated through 2 evaluation studies, one in the academy and the other in industry. These studies provided evidences that ArqCheck can be used to detect defects in architectural models. Therefore, in this study only the questions related to Reusability, which were added to the checklist in our work, were applied to the architectural model. The goal of the study can be stated as follows: **Analyze** the extensions performed in ArqCheck to evaluate architectural elements Reusability; **For the purpose of** characterization; **With respect to** the feasibility of detecting architectural defects concerning Reusability; **From the point of view of** software inspectors; **In the context of** the inspection of an architectural documentation produced in the same academic environment as this work, which encompasses the required requirements for the application of the checklist.

The object of the study was an application developed in the same academic environment as this work. It contains minimum requirements for the application of the checklist, such as: the system has a requirements documentation and they are traced to the architecture. The inspectors were software engineering graduate students from different research groups, who had classes and, in general, some experience in software inspection. Five inspectors were available to participate in the study. Two of them had experience in software reuse methods and the others had low experience. They were trained in using ArqCheck and received the necessary documentation for the inspection.

At the end, an inspection meeting was conducted among the inspectors, the researcher who made the extensions in the checklist as the moderator, and 2 system experts who judge when an identified discrepancy was a defect or not. Before this meeting, the researcher in the role of the moderator, made the consistency among the discrepancies identified by the inspectors. After this meeting, it was realized that 76% of the discrepancies identified were architectural defects in fact and 65% were defects concerning Reusability. All the inspectors identified more Reusability defects than other kinds of defects or false-positives (i.e. discrepancies that do not represent defects). In general, the inspectors agreed that the checklist helped in identifying architectural defects concerning Reusability, although some improvements were suggested in the checklist: some questions were redundant and should be transformed in a unique question; some questions were subjective and did not help in identifying defects; one of the questions was not used for the identification of Reusability defects and needed a review.

Therefore, after this study the checklist was reviewed and applied again in the last evaluation study detailed in the next Section. It was also interesting to note that domain knowledge is important for the success of an inspection, since the inspector who identified more defects was the one that had the greatest domain knowledge. Moreover, reuse knowledge was also important since the 3 inspectors who had higher reuse knowledge achieved the best performance in the study.

4.4. Final Evaluation Study: ArchMine integrated with ArqCheck

4.4.1. Evaluation Study Goals and Definition

Along the previous studies, it was possible to get some evidence about ArchMine and ArqCheck feasibility. However, an evaluation study within an industrial context is important in order to assure the feasibility of the approach in a real scenario. Moreover, ArchMine had not been previously evaluated together with ArqCheck. Therefore, the goal of this final evaluation study was to evaluate if the extended version of ArqCheck, when incorporated to ArchMine, could reduce architecture evaluation effort and improved the architectural quality for reuse. The evaluation study is outlined as follows: **Analyze** the evaluation of the architectures recovered by ArchMine through the extended version of ArqCheck; **For the purpose of** characterization; **With respect to** architecture evaluation effort reduction and architecture reusability improvement; **From the point of view** of software engineers; **In the context of** the recovery of an object oriented framework, written in Java, and in use in an industrial context.

4.4.2. Evaluation Study Hypothesis, Metrics, and Threats to Validity

This evaluation study intended to reject 2 null hypotheses, namely: *($H0_1$)* – the incorporation of an extended version of ArqCheck, to evaluate the architectures recovered by ArchMine, does not reduce the evaluation effort; and *($H0_2$)* – the quality of the recovered architecture, concerning Reusability, is not improved by incorporating an extended version of ArqCheck to architecture evaluation.

In order to test these hypotheses, some metrics were established, namely: M1 - average time spent to architecture evaluation by the inspectors; M2 - number of inspectors who found extended ArqCheck difficult to apply; M3 – number of inspectors who found that the quality of the architecture, concerning reuse, had improved, which means that the recovered elements were cohesive and implemented a domain concept. Metrics M2 and M3 were derived by counting the answers given by the inspectors in a post-evaluation form. Some threats to the study validity were also identified. Only 1 application was used in this study, which can represent a confounding factor due to its specific characteristics. But in the industry it is difficult to involve many applications implying in the allocation of too much time of system experts. In order to evaluate the reduction of the effort in architecture evaluation, it would be necessary to evaluate the same architecture using ArqCheck and an ad-hoc, or any other evaluation method. However, there were not enough available experts to perform these evaluations. Architecture evaluation performance was then compared against the one obtained in the previous study.

4.4.3. Evaluation Study Execution

The architecture of the CSBase framework (LIMA *et al.*, 2006), that manages resources in a distributed grid environment and was developed in a partnership between academy and industry, was recovered following the heuristics defined in ArchMine (Section 3.1.3). It has 720 classes. Use case scenarios were defined by the researcher with a system expert, following ArchMine guidelines, in a total of 129.

After architecture recovery, the inspection process took place. The researcher, in the role of moderator and document author, planned the inspection process by configuring the checklist to evaluate CSBase recovered architecture. Although it is the moderator's responsibility to choose the inspectors, in this special case the inspectors were selected by the CSBase manager. The checklist was configured by classifying the questions in applicable or not, changing some architectural terms to the adopted architectural representation, and by instantiating the quality scenario for the non-functional Reusability requirement. Table 4 presents an excerpt of the checklist used in this study. In order to classify the questions in applicable or not, in some cases, the moderator needed the help of the system experts. Question 8, for example, was not applicable in this study because CSBase did not have a previous documented requirements model.

After configuring the checklist, the moderator presented the recovered architecture and the checklist to the inspectors, training them in applying ArqCheck. They were also given a discrepancies report, to be filled whenever they found that an answer in the checklist indicated a defect in the architecture. Inspectors were asked to fill up some evaluation forms informing the difficulty degree in applying ArqCheck and the quality of the architecture for reuse, before and after correction. Once the inspectors concluded the discrepancies detection, the moderator consolidated the discrepancies list by elimi-

nating redundancies. The final defects list was generated after the inspection meeting. Based on this list, the moderator conducted the rework activity, contacting system experts whenever he had difficulties to correct the architecture. The restructured architecture was presented to the system experts (i.e. inspectors), who evaluated its final quality.

Table 4. Excerpt of the checklist used in the evaluation study

Nº	**Items that Evaluate Architectural Representation Consistency**	Yes	No	NA
1.	In the diagrams, is there any architectural element that does not have relationships, being isolated from the others?			
2.	All the architectural elements, identified by their names, were represented by the same abstraction in the different diagrams?			
Nº	**Items that Evaluate the Conformance to Functional Requirements**	Yes	No	NA
8.	All the functional requirements, or quality attributes or other requirements, created by the architectural decisions, is being satisfied by any architectural element?			x
Nº	**Items that Evaluate the Conformance to Non-Functional Requirements (Reusability)**	Yes	No	NA
9.	The responsibilities of the internal modules (i.e. classes) of a reusable element belong to the same context, i.e. they intend to achieve the same goal or are used in the same use case scenarios?			
10.	Is it possible to identify groups of reusable architectural elements with similar responsibilities or that share some common implemented functionalities that should be grouped to compose a component?			
11.	From the point of view of the concept that the reusable architectural element represents, are there modules (i.e. classes) that should be allocated in it, considering their responsibilities or functionalities, but that are allocated in another architectural element?			
12.	Are there couplings between a reusable architectural element and other elements that hinder its reuse?			
13.	Considering the coupling among reusable architectural elements, are there couplings that justify their clustering in one component?			

4.4.4. Evaluation Study Results and Lessons Learned

Concerning the null hypotheses, there were evidences to reject the second one. Concerning $H0_1$, architecture evaluation effort was almost not reduced in comparison to our previous studies. It was close to the one spent by the system expert of Odyssey in the second study, as can be seen in the values presented in Table 5. There was a small reduction in the time spent for each class evaluation. However, in order to test $H0_1$ hypothesis it is necessary to perform new studies with the same application being evaluated by 2 methods (i.e. ad-hoc and ArqCheck). Concerning the second hypothesis $H0_2$, the system inspectors agreed that ArchMine recovered architectural elements representing domain concepts for CSBase and that the quality of the architecture for reuse had improved after inspection, according to the values of metric M3.

Table 5. Architecture evaluation effort

Application	Classes	Architecture Evaluation Effort	Effort Evaluation per Class
Odyssey	595	5 hours	≡ 1,9 minutes
CSBase	720	7 hours	≡ 1,7 minutes

Some lessons learned were: CSBase is client-server and since the client and the server were separately monitored, some interface components were not adequately captured; architectural element names, in general, adequately reflected CSBase domain concepts, which indicates that names derivation strategy is a good contribution of ArchMine when compared to other reverse engineering approaches (e.g. Bunch (MITCHELL & MANCORIDIS, 2006)); minimum confidence was 60%, the value that was tuned along the previous studies; the great effort to apply ArqCheck is also due to

limitations of the Odyssey environment, where the architecture is recovered, with respect to the manipulation of the recovered models. For instance, Odyssey does not allow to query for a class and to count the classes in an architectural element.

5. Conclusions

In this paper we presented 4 evaluation studies that were conducted to evaluate ArchMine and its corresponding tool set. As result, it was possible to get the following evidences concerning the proposed approach goals: ArchMine can recover architectural elements that represent domain concepts; the approach can be used across different domains and implementations; ArchMine can help in getting an architectural model that can be used for reuse and program comprehension.

However, the evaluation studies also allowed to identify limitations of the approach. Minimum support and confidence, antecedents selection, and specified use case scenarios have a great impact in the recovered architecture. These limitations are minimized by the heuristics and guidelines proposed by the approach, which were refined along the evaluation studies. These can be continuously refined as the approach is applied in new case studies, although the current ones proved to provide good results within the analyzed studies. Moreover, a critical point in the approach is use case scenarios definition. It is clear that varying input data or paths executed in the application impacts the methods and classes that are exercised. Along the evaluation studies it was observed that with a reasonable sample of use case scenarios it is possible to recover a valuable architectural model for the application. As future work, divergences in the recovered architectures by varying input values in the process may be evaluated. Moreover, dynamic view reconstruction must also be evaluated through evaluation studies.

6. References

AGRAWAL, R., SRIKANT, R., 1994, "Fast Algorithms for Mining Association Rules". *20th Very Large Databases Conference*, pp. 487-499, Santiago, Chile, September.

ANQUETIL, N., FOURRIER, C., LETHBRIDGE, T.C., 1999, "Experiments with Hierarchical Clustering Algorithms as Software Remodularization Methods". In: *Working Conference on Reverse Engineering*, pp. 235-255, Pittsburgh, PA, USA, October.

ANQUETIL, N., LETHBRIDGE, T., 1999, "Recovering Software Architecture from the Names of Source Files", *Journal of Software Maintenance: Research and Practice, John Wiley & Sons Ltd.*, v. 11, pp. 201-221.

BARCELOS, R., TRAVASSOS, G.H., 2006, "Evaluation Approaches for Software Architectural Documents: A Systematic Review". In: *Ideas 2006*, v. 1, pp. 433-446, La Plata, Argentina, April.

BASS, L., CLEMENTS, P., KAZMAN, R., 2003, *Software Architecture in Practice*, 2nd ed., Addison-Wesley.

BOJIC, D., VELASEVIC, D., 2000, "A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering". In: *4th CSMR*, pp. 23-31, Zuriq, Swiss, February/March.

CORREA, A.L., WERNER, C.M.L., 2004, "Applying Refactoring Techniques to UML/OCL Models". In: *International Conference on the Unified Modeling Language (UML'04)*, pp. 173-187, Lisbon, Portugal, October.

DANTAS, A.R., *et al.*, 2006, "Model Driven Game Development - Experience and

Model Enhacements in Software Project Management Education", *Software Process Improvement and Practice*, v. 11, n. 4, pp. 411-421.

DEURSEN, A.V., HOFMEISTER, C., KOSCHKE, R*., et al.*, 2004, "Symphony: View-Driven Software Architecture Reconstruction". In: *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pp. 122-132, Oslo, Norway, June.

DUENAS, J.C., DE OLIVEIRA, W.L., DE LA PUENTE, J.A., 1998, "A Software Architecture Evaluation Model". In: *2nd International ESPRIT ARES Workshop*, pp. 148-157, Las Palmas de Gran Canaria, Spain, February.

KAZMAN, R., BASS, L., G., A*., et al.*, 1994, "SAAM: a Method for Analyzing the Properties of Software Architectures". In: *International Conference on Software Engineering (ICSE)*, pp. 81-90, Sorrento, Italy, May.

KAZMAN, R., CARRIÈRE, S.J., 1997, *Playing Detective: Reconstructing Software Architecture from Available Evidence*, Technical Report CMU/SEI-97-TR-010.

KAZMAN, R., KLEIN, M., CLEMENTS, P., 2000, *ATAM: Method for Architecture Evaluation*, CMU/SEI, Technical Report, CMU/SEI-2000-TR-004.

LIMA, M.J.D., URURAHY, C., *et al.*, 2006, "CSBase: A Framework for Building Customized Grid Environments". In: *Third International Workshop on Emerging Technologies for Next-generation GRID*, pp. 187-192, Manchester, UK, January.

MITCHELL, B.S., MANCORIDIS, S., 2006, "On the Automatic Modularization of Software Systems Using the Bunch Tool", *IEEE Transactions on Software Engineering*, v. 32, n. 3 (March), pp. 193-208.

MURTA, L.G.P., *et al.* 2006, "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links". In: *International Conference on Automated Software Engineering (ASE)*, pp. 135-144, Tokyo, Japan, September.

O'BRIEN, L., STOERMER, C., 2003, *Architecture Reconstruction Case Study*, Software Engineering Institute, Technical Note CMU/SEI-2003-TN-008.

ODYSSEY, 2007, "Odyssey: Software Reuse Infrastructure based on Domain Models". In: http://reuse.cos.ufrj.br/site/en/, accessed in May.

SAMETINGER, J., 1997, *Software Engineering with Reusable Components*, Springer-Verlag New York, Inc.

SARTIPI, K., 2003, *Software Architecture Recovery based on Pattern Matching*, PhD Thesis, School of Computer Science, University of Waterloo.

SCHMERL, B., *et al.*, 2006, "Discovering Architectures from Running Systems", *IEEE Transactions on Software Engineering*, v. 32, n. 7 (July), pp. 454-466.

TZERPOS, V., 2001, *Comprehension-Driven Software Clustering*, PhD, Department of Computer Science, University of Toronto.

VASCONCELOS, A.P.V., 2007, *An Approach to Support the Creation of Domain Reference Architectures based on Legacy Systems Analysis*, PhD Thesis, COPPE, UFRJ, Rio de Janeiro, April, In Portuguese.

VASCONCELOS, A.P.V., WERNER, C.M.L., 2007a, "Architectural Elements Recovery and Quality Evaluation to Assist in Reference Architectures Specification". In: *19th SEKE Conference (SEKE'2007)*, pp. 494-499, Boston, USA, July.

VASCONCELOS, A.P.V., WERNER, C.M.L., 2007b, "Architecture Recovery and Evaluation Aiming at Program Understanding and Reuse". In: *Third International Conference on the Quality of Software Architecture*, pp. 65-82, Boston, USA, July.

WOHLIN, C., RUNESON, P., HÖST, M*., et al.*, 2000, *Experimentation in Software Engineering: an Introduction*, USA, Kluwer Academic Publishers.