A Lightweight Process for Architecture Recovery: From Code to Domain Requirements and Back Again

Davor Svetinovic and Michael Godfrey
School of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1, CANADA
{dsvetinovic, migod}@uwaterloo.ca

ABSTRACT

For many information systems, both the problem domain and the supporting computing infrastructure change over time. As new features are added, new environments are supported, and old defects are fixed, the cumulative effects of these maintenance activities often pull the design elements of the system in different directions, causing architectural drift, conceptual inconsistencies, and a widening of the gap between requirements and code. Explicitly modeling the software architecture of a system as a part of the maintenance process can aid in lessening the negative side effects of maintenance, as the software architecture model serves as a partial bridge between the requirements of the business domain and the source code. In this paper, we present a lightweight process for architecture recovery that aids developers in creating and maintaining software architecture models. The process is designed to be practical for the recovery of architectures of small to mid-sized software systems; it is based on and extends the PBS tool architecture recovery approach and goal-based requirements engineering theory.

1. INTRODUCTION

Since its early days, software development has been implementation driven. Programming, still considered by many as the most important and difficult development activity, has attracted most of the research attention over time. While sufficient in some cases, programming has become a relatively routine activity compared to the other development activities in the development of today's large, complex, and constantly changing software systems. The main difficulty in today's development is not anymore *how* to build the system, but *what* to build and how to make it as adaptable to future change as possible [7].

Because of its early importance, implementation technologies and paradigms have influenced all development stages, even the early ones such as requirements analysis and design. For example, structured and object-oriented programming paradigms resulted in structured [10, 34, 33] and object-oriented analysis and design techniques [19, 8, 6]. This tradition continues with emergence of new methodologies such as aspect-oriented analysis [1], which has its origin in aspect-oriented programming paradigm.

The success of such approaches was mostly due to the fact that the traditional way of development focused on one product at a time [17]. A clear product-level requirements specification combined with low-level design, using structured or object-oriented concepts, was appropriate for a product development in relatively stable and well-understood problem domains.

Domain-level requirements analysis and specification appeared as a solution to a need for building software systems for large, difficult to understand, and changing problem domains. Goal-driven requirements engineering emerged as a leading approach for dealing with domain-level requirements for large systems [9, 20, 31]. The main emphasis of this approach was on making sure that software actually fulfills business goals. This goal fulfillment problem was one of the main weaknesses of the traditional product-level requirements engineering approach.

Now, with the emergence of new economic trends, the Internet as a business medium, software as a commodity, *etc.*, even small systems have become much more difficult to build and maintain. New software paradigms and technologies such as web services, agility, and product lines, emerged to solve this new wave of problems. In this new situation, both business systems and software systems change faster than ever before. Naturally, both domain and product-level requirements specifications become obsolete very quickly, in some cases even before the product is built [13, 17]. This problem, in addition to the old problem of legacy systems, raises a need for an efficient domain artifacts recovery process. We have been working on an lightweight architecture recovery process that can be used for the recovery and integration of both, domain and system architectures.

2. THE PROBLEM AND OUR APPROACH

In a situation where both problem domain and supporting computer system constantly change, and where one system is used to support multiple different problem domains and

has to be adaptable to the new ones, we need an efficient way for recovering and preserving artifacts at both, domain and system levels. Software architecture is considered to be one of these artifacts.

The goal of our research was to develop a systematic approach for recovery and presentation of a software architecture and its rationale (*i.e.*, the underlying reasons for that particular architecture) for small to mid-sized systems. Our goal was not to invent yet another architecture recovery process, but to build on existing ones and to conform to lightweight development philosophy. Two main properties of target systems are their moderate size and changing requirements in their problem domain. Also, the particular emphasis was on the recovery of the architecture rationale, as design decisions made during development and requirements that led to them were not documented and preserved. Architecture rationale recovery provided us with answers to why architecture was designed or has drifted in a particular way.

It was not the goal of our research to develop the definitive set of activities to be performed and artifacts to be built. This would be contrary to the inherent difficulties of software development, properties of different problem domains, different circumstances under which recovery is performed, and purposes for which artifacts are produced. The goal was to lay out a theoretical background for the recovery process and clearly define the constraints to be satisfied. This was done through the use of different meta-models, such as problem domain and architecture metamodels, and the analysis of the conformance of different artifacts to the properties imposed by the lightweight constraints. We applied the approach for architecture recovery of a mid-sized software application to evaluate its effectiveness, usability, and applicability.

3. BACKGROUND AND RELATED WORK

In this section we present background and related research on software architecture, design, and attribute theory, emphasizing how it relates to our research that is presented in the rest of this paper.

3.1 Software Architecture and Design

One of the main tools that has been used in all areas of software engineering to solve different problems is abstraction. Software architecture is a recent significant result of its use. It helps engineers to conquer the complexity of the system by giving them the ability to organize, understand, present, and manage system in a much more natural and easier way, than they can just do by using programming language features.

Currently, there is no universally accepted definition of what software architecture is. Nevertheless, common to all proposed definitions is that architecture deals with large scale constructional ideas and techniques — high level organization of system, constraints, motivation and rationale, architectural styles, interconnections, and collaborations [22, 26, 4].

Software architecture research can be categorized in following groups, which together form a body of knowledge and techniques that software architects use during development:

- Architectural styles and design — study of proven architectural designs and practices that help architects design flexible and extensible software systems.
- Component-based technologies — different technologies that are used to build systems using proven architectural practices.
- Architectural recovery, visualization, and analysis — the focus of this research is on increasing system's understandability, reliability, and reusability.
- Run-time system generation and manipulation — the study of different techniques for run-time system modularization and modifications.

The focus of our research is on architectural recovery, visualization, and analysis. The following architectural topics and techniques are the main tools used by researchers in this area:

- Architectural views
- Reference architectures
- Architectural refactorings and repair
- Architecture reengineering and visualization

3.1.1 Architectural Views

The biggest problem in trying to determine how best to represent the architecture and design of a software system is that there exist many distinct concerns. It is very hard to present all of them using a single diagram, and even if one succeeds, the usefulness of that diagram is doubtful. Thus it is common to separate these different concerns.

Kruchten's landmark paper on architectural views considers that there are "4+1" key architectural views [18]:

- Logical View — describes the conceptual decomposition of system into modules, layers, and other architectural constructs.
- Process View — describes the run time decomposition of system into processes and emphasizes concurrency and synchronization aspects.
- Physical View — describes the mapping of software components onto hardware components and emphasizes distributed aspects of the system.
- Development View — describes the source code and other development modules organization in a development environment.
- Use-Case View — this view unifies all the previously mentioned views through the use of use-cases.

Each of these views presents two kinds of system properties, static and dynamic. Notation used to capture these views should provide support for both properties.

3.1.2 Reference Architecture

A reference architecture is the common architecture model for a family of products. It represents the structure that is commonly found, possibly with slight variations, in products that belong to that family. There are some well-known reference architectures, for example, for compilers [2, 26] and operating systems [28]. The main purpose of creating a reference architecture is to codify knowledge about a particular product family, and to serve as a starting point in building a new product. It can also used as a guide to control the development and prevent anarchy in the system, usually by reengineering the system to fit its reference architecture more closely.

3.1.3 Architectural Refactorings and Repair

Refactorings are changes made to the code that improve its design in some way. Continual refactoring is an important feature of lightweight methodologies, as it helps to compensate for the lack of upfront design activities. A catalog of refactorings can be found at [23], and a definitive guide to performing them is [11].

Of particular interest to us are the refactorings that affect the architecture of the system. Some of them are presented in [30], together with a few successful applications. For example, architectural refactoring is a process of separating the user-interface functionality from the application logic in software systems where these two separate concerns were merged together.

The current problem with existing refactorings is that they are not automated and are tedious to perform manually. More robust support for automatic refactoring has to be developed and tightly integrated with the other development tools — one example of such a tool is refactoring support integrated in IntelliJ IDEA integrated development environment [14].

3.1.4 Architecture Reengineering and Visualization

One of the main uses of the software architecture is to help one understand the system. There have been many efforts recently to automate architectural reengineering and to present this information visually. Most of these tools have been used exclusively for the visualization of the static structure of the system and interconnections between its components. Examples of these kinds of systems include PBS [21], Rigi [24], and SHriMP [27].

The usefulness of such systems is in their ability to present us with important architectural issues, while hiding lower-level design decisions. For example, the PBS system allows us to analyze a software system at different levels of abstraction, with the lowest level being the source file level. It also allows us to analyze the dependencies between different subsystems, and the dependencies propagation through function calls, variable references, etc.

3.2 Quality Attribute Theory

Quality attributes are the descriptions and specifications of the characteristics of a software system and its ability to fulfill its intended functionality, while satisfying all the imposed constraints. The study and use of quality attributes has made many contributions to software engineering practice. Goal-oriented requirement engineering processes have helped capture and model a wider range of requirements than previously possible, improve requirements traceability, and facilitate the process in general [20]. Attribute-Based Architecture Styles (ABAS) have allowed qualitative reasoning about the use of a particular architectural style [16]. Architecture Tradeoff Analysis (ATA) method relies upon the use of quality attributes to analyze and express the architectural tradeoffs [15]. Quality attributes are the initial artifact for several architecture design processes, including the Architecture-Based Design Method [3]. Besides providing the driving force, quality attributes also serve as the connection among all these techniques and methods [12].

All of the previously mentioned techniques are based upon a solid understanding of the interdependencies among attributes, especially conflicting ones. Several studies have tackled the problems of complex dependencies between the attributes and how to manage them [5, 31].

4. DOMAIN ARCHITECTURE RECOVERY

In this section, we present our architecture recovery process. In the first part of this section, we discuss architecture recovery issues and problems, including techniques used to solve them. In the second part, we present our architecture recovery process in a development process neutral way. In the third part, we discuss possible ways of adapting and using our process.

4.1 Architecture Recovery Issues and Goals

Most software systems are built on top of a pre-existing code base, such as legacy applications and large libraries. Many of these applications are built without the careful use of forward design practices, and for most of them, their structure is not documented or the documents are obsolete. As qualities of our new application will depend largely on qualities of legacy systems used, we need a way of recovering and modifying the architecture of these legacy applications. Also, processes that do not emphasize forward design, and requirements that constantly change make it even harder to achieve major system goals.

The main goals of a successful architecture recovery process are to produce artifacts that describe:

1. the actual architecture of the business domain and computer system under consideration, and
2. the rationale of that architecture — why architecture is as it is.

The second goal is considerably harder to achieve since it includes recovery of forward design decisions made, external influences that produced them, and alternatives that were considered and why they were not implemented. The aim of our approach is to address both of these goals. In addition, in order to be an efficient and lightweight approach, our process aims to achieve the following goals and constraints: minimal additional developer's training, low risk incorporation in the development process, robust roundtrip tool support, minimal and simple set of artifacts directly

usable later for forward engineering, and use of the forward engineering principles to recover architecture rationale.

4.2 Architecture Recovery Process

Ideally, the most influential force that drives architectural design should be the system's functional requirements and quality attributes. While often this is the main force that shapes the architecture of a system, there are many others: different technical, business, and people-oriented influences shape architecture in both positive and negative ways. For example, in a company that has development teams at different locations, actual work division can essentially dictate the architecture of a system.

The fact that architecture is primarily derived from and influenced by quality attributes in the context of functional requirements imposes that an architecture recovery process should trace from the concrete architecture of the system back to the actual requirements that originally shaped it. At the same time, the analyst that is performing architecture recovery must isolate other influences and their effects on software architecture. Therefore, the main groups of activities performed during architecture recovery are:

- discovery of concrete architecture of the system;
- discovery of functional requirements and quality attributes as a major force behind architectural decisions;
- recovery of architecture design decisions that have led to actual concrete architecture of the system, and identification of other possible architectural solutions and their advantages and drawbacks; and
- identification of other factors that have influenced the architecture.

The first activity is physical architecture recovery, and last three concern architecture rationale recovery, which together make a complete set of architectural artifacts.

We define our process in three steps. In the first one, we introduce major concepts that participate in the process, including sources of information, different domains of concern, different stakeholders, etc. We relate them using our architecture meta-model, which is used to keep our process focused and consistent. In the second part, we introduce the techniques used to manipulate these concepts, and architecture recovery artifacts that are produced as the result of application of these techniques. In the last part, we introduce the process steps that relate these different techniques, and provide guidelines on how to perform them.

4.2.1 Architecture Meta-model

In this section, we introduce the essential concepts that occur in an architecture recovery process. Architecture recovery spans several domains of concern, and an analyst has to capture all concepts and relations that occur in these domains. In order for a process to be focused, and architecture recovery efficient and useful, we have to choose a manageable subset of these concerns in a way that will maximize the benefits of capturing and documenting them.

The first major division of these concerns is on:

- business domain architecture, and
- computer system architecture.

Business Domain Architecture

A computer system is a part of a larger business system, and serves as a resource to accomplish certain business goals. During an architecture recovery process, we need to study different aspects of the business domain in order to understand our software system. A large amount of information about business architecture is gathered, but often much of that information is lost and not documented during reverse engineering processes that do not value that type of information. Our recovery process tries to capture and preserve this information as it is very valuable for long term development goals, and is a major source of architectural influences.

There are four main sets of concepts that describe business architecture:

1. *Business Resources* — All entities, both physical and abstract, that exist inside a business environment. These include people, information, different computer systems, business supplies and products, etc. These are entities that participate in business processes. A subset of these resources is a source of modelling concepts for software systems built using object-oriented or component-based methodologies. The value of tracking and preserving knowledge about these concepts is in the fact that they serve as a tool to perform analysis of our software system architecture, and to track changes to business and software system since when system was built as a part of evolution study and to evaluate how well a software system reflects today's business needs. A computer system for which we are performing architecture recovery is a resource within one or more business ecosystems.
2. *Business Goals* — The purpose of performing a business activity is to achieve certain goals. Goals can be decomposed into subgoals, and at a certain level of decomposition, we reach business goals that have to be satisfied directly by our software system and its architecture. The study of business goals allows us to evaluate how well our software system's goals conform to them, and how we should improve our system. An important part of this work is the study of the evolution of these goals so the architecture of our system will be able to support future goals and to remove obsolete ones.
3. *Business Processes* — A system for which we are recovering architecture may participate within several business processes in order to help achieve certain business goals. The most common forward software analysis technique is discovery of use-cases to capture requirements. These use-cases describe sub-processes of larger business processes that are automated by our system. It is important to understand business processes as they relate all the use-cases of our system,

which in turn relate requirements that our system has to satisfy. This allows us to study the architecture of a system within a context and to analyze the architecture.

4. *Business Rules* — Business rules are a major source of constraints on software system. Many of these constraints directly influence software architecture. As such, it is important to understand them and keep track of them, for example, to remove architectural limitations imposed by constraints that do not hold any more.

Computer System Architecture

Many software development technologies, such as object-oriented development technology, allow us to simulate concepts that exist in a problem domain. For our purposes, we define the following architecture concepts that we will keep track of:

- system,
- subsystems,
- modules,
- connectors,
- processes,
- logical processes, and
- hardware devices.

System concept defines the outermost boundary of the software system under consideration. Same as for all other concepts, the amount of information and its scope depends on the architecture view in which it is used. It serves as a container for all other concepts, and defines the computer system as a resource in the business model.

Subsystems are abstract concepts that serve as abstraction tools for management and abstraction of actual physical modules, connectors, and processes. They serve as containers and building blocks of the whole system. Depending on the architecture view, they capture different concerns of the system.

Modules are basic architectural building blocks. For example, in the logical view they represent concepts that occur in business domain, and in the implementation view they represent code units. Modules are abstractions of basic building blocks of the system, depending on the development technology used.

Connectors are abstractions of communication mechanisms and channels that exist in a system. Their size and complexity vary from simple procedure calls to connectors built from several modules and hardware devices.

Processes are physical, run-time processes that perform activities that fulfill goals of logical processes. They are allocated to possibly many different processing nodes, and are basic building blocks of run-time view.

Logical processes are white-box use-cases. The difference between them and regular, black-box, use-cases is that they include descriptions of which activities are performed within the system.

Hardware devices are concepts that occur in run-view, and represent actual hardware device that are parts of a computer system.

Figure 1 shows the main concepts, and main relationships at the architectural level. This is not the only possible decomposition, but is a useful, minimal one that will help us focus on the main architecture issues without getting lost in many details, which are usually not necessary. Simplicity of the decomposition is also in the spirit of our lightweight approach, and allows us to abstract from the details of used low-level development technology.

Now we will present techniques that are used to recover and present these concepts in order to provide a useful architectural description.

4.2.2 Architecture Recovery Techniques

Before we embark into the discussion of particular architecture recovery techniques and artifacts, we summarize the main goals of a successful recovery. These goals will be used to classify and relate specific techniques. These goals are:

- Discovery of the current structure of the system.
- Discovery of the main influences that have led to that architecture — most important are quality attributes (in context of functional requirements).
- Discovery of the design decisions that have led to the current architecture of the system and possible alternatives.

Discovery of Present Architecture

Our approach to the recovery is iterative and view-driven. Architectural views provide separate conceptual approaches that allow us to focus on recovery of only a portion of architectural concerns and facts at a time. Iterative means that we do not focus on completion of architecture recovery from only one perspective (view) at a time, but iterate among recovery from different perspectives and use feedback from one to improve another.

The main work flow is:

1. Initial recovery from problem domain perspective.
2. Recovery from logical, run-time, and test perspective.
3. Recovery from problem domain perspective, and iteration to step 2.

One of the main goals of a good architecture is to preserve problem domain concepts in software architecture. Therefore, during recovery we pay special attention to discovery of actual mappings from problem domain view to other views,

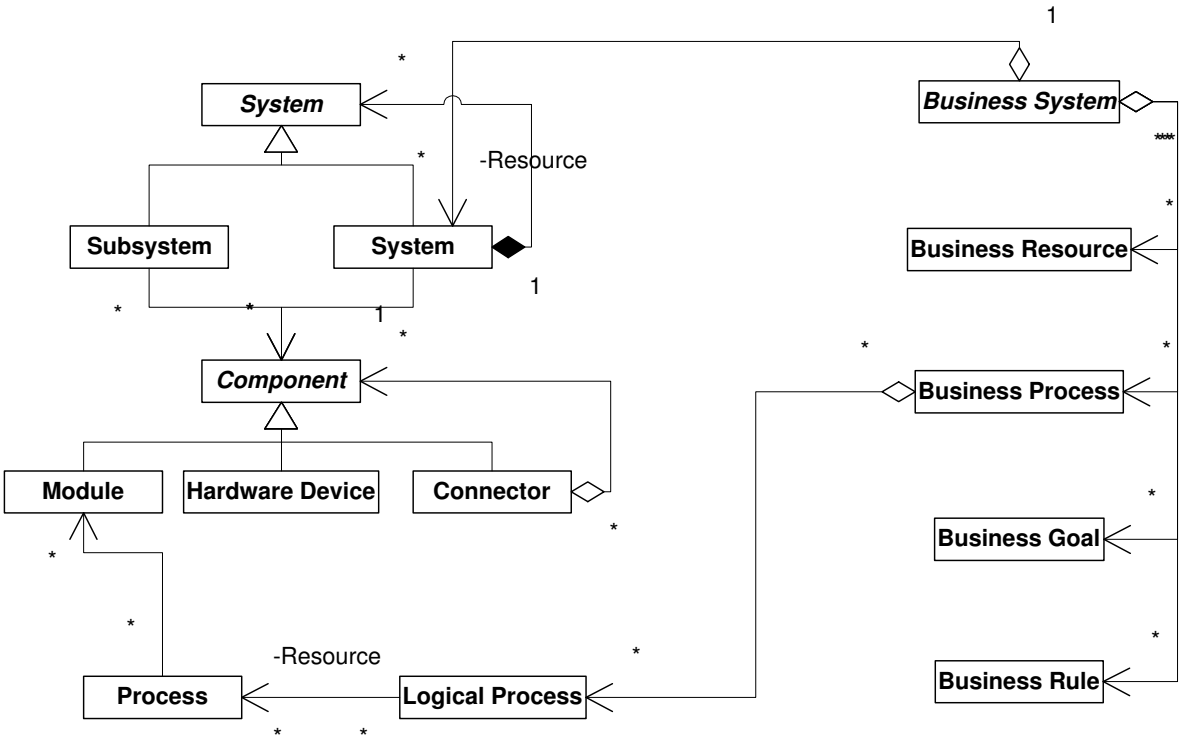

Figure 1: Architecture Meta-model

and vice-versa. This is done by tracking which concepts from one view allow discovery of concepts in another. Only architecture-relevant concept dependencies are recorded.

Problem Domain View

In order to perform architecture recovery from a problem domain perspective, we use forward requirements and business engineering techniques such as employee interviews. The focus is on discovery of business resources, goals, processes, and rules that are closely related to our software system.

Two possible approaches that we are using, depending on the type of system under consideration, are actor-driven and process-driven approaches. The actor-driven approach is useful for systems where actors, as active resources, control processes that occur inside the system. For example, in business environments, employees as active resources control most of the business processes, so for recovery of information systems from a problem domain view perspective, we should use an actor-driven approach. The main steps are:

1. Discover an actor.
2. Discover the goals of that actor.
3. Discover the processes performed to achieve these goals and the resources used or affected by them.

On the other hand, the process-driven approach guides recovery more effectively in business domains where actors are of secondary importance. For example, for an industrial control system problem domain, the main focus is on processes that occur within it and actors are of secondary importance. The main steps are:

1. Discover a process and the resources used.
2. Discover which goals are achieved by that process.
3. Abstract goals at the level of business goals.
4. Identify which resources are actors.

Contrary to business engineering needs, for our purposes we do not require a large amount of detail. The main artifacts produced are:

- Actor-Goal List
- Goal-Process List
- Process-Resource List
- Business Rules List

The main purpose of these artifacts is to preserve knowledge about identified concepts and relationships among them. We use a simple list of items although other formats, such as UML diagrams, are possible. The main reason for using

simple lists is to minimize the effort in producing and updating these artifacts.

Logical, Run-Time, and Test Perspective

The most important perspective for our purposes is the logical view of the system. It serves as a bridge between the problem domain perspective and the implementation perspective. It also has a strong influence on the run-time and test aspects of the system.

Logical view is composed of a set of concepts, some of which appear in the pure problem domain, and some in a pure implementation view. The main value of this view lies in the fact that it provides a focus and is the main source of information for actual development of the software system.

This view is further subdivided in two different aspects, which occur naturally during the recovery process:

1. conceptual architecture, and
2. concrete architecture.

Conceptual architecture is a very high level view of a system's architecture, which occurs often as a transition step when one analyzes first architecture from a problem domain perspective and then moves toward analysis at the implementation level. Concepts that appear in artifacts presented at the conceptual architecture level are largely a subset of concepts that occur in the problem domain view.

On the other hand, concrete architecture is a more detailed view of the system and appears when one is analyzing a system using the bottom-up approach (*i.e.,* starting from source code and then abstracting concepts and mapping them to the concepts that occur in the problem-domain view). Concepts that appear in artifacts at this abstraction level are largely a subset of concepts from pure implementation view.

Our approach is mixed and encourages iteration in the refinement of artifacts at both abstraction levels. As conceptual architecture is closer to problem domain concerns, it is recovered using familiar object-oriented, or component-based analysis techniques and general problem domain knowledge. Concrete architecture is, on the other hand, recovered using reverse engineering tools like source code browsers. Finally, a mapping between these two abstraction levels are established, with special emphasis on mismatches, which are potential candidates for architectural refactorings.

Concepts that appear in this view are:

- system,
- subsystems,
- modules,
- connectors, and
- logical processes.

The main techniques used to produce artifacts in this view are:

- reference architecture, and
- responsibility based dynamic architecture representation (architectural use-cases).

The Run-Time view in our process is limited to only a basic description of computer system hardware and process distribution. Only main processes are discovered, presented on a diagram, and mapped to their processing nodes. The only purpose of this diagram is to allow us to map the main sets of functionality of our system to distributed aspects of business domain. We do not study it in more detail as the run-time view depends highly on the choice of hardware and middleware technologies, and the whole purpose of architectural recovery is to abstract above these low-level concerns. More detailed study of run-time aspects of a system is appropriate during low-level design recovery of a system.

Test view is an optional view. It is mainly used when a system has an existing, extensive set of automated, software-based testing artifacts such as unit tests. Techniques used to recover testing aspects of a system are the same as the ones used for recovery from logical perspective, so we will not discuss them again. The main use of test view is in conjunction with logical view to provide a more complete mapping to problem domain view. The test view is especially important when studying the evolution of a software system, as it represents current external needs from business perspective (*i.e.,* requirements that our system as a resource is supposed to fulfill).

Now, when we know the state of the current architecture of the system, we need to find its rationale.

Quality Attributes and Design Decisions Recovery

Similar to recovery of architecture from the logical view perspective, the discovery of quality attributes is a two way process. In one direction, quality attribute recovery is a requirements engineering activity where one analyzes a business system to find out which qualities a computer system as a business resource has to have and fulfill. In the other direction, quality attribute recovery is a technique performed in conjunction with design decision recovery, and basically represents an answer to a question of why a particular design decision has been made.

Quality attribute discovery from requirements perspective is a well studied activity, described in many requirements engineering texts [25]. Compared to the actual requirements engineering process, ours is simplified since we concentrate only on discovery of architecture-relevant quality attributes. As a main guideline, we have an initial set of commonly occurring quality attributes for a particular problem domain (there are many classifications of quality attributes) such as reliability, performance, etc. For each one of these high-level quality attributes we are discovering, analyzing, and documenting *quality facts* — simple statements that quantify these quality attributes in the context of functional requirements. For example, "in a case of main system failure, the

monitoring system must restart the main system within 10 seconds". These quality facts are documented in *quality attributes table*, and are related directly to high-level quality attributes, and functional requirements within which they are analyzed.

Quality attribute recovery from the other perspective is guided by the analyst's expertise in forward architecture design techniques. Many architecture styles today are directly connected to a set of attributes for which they are particularly well suited [16]. The analyst's role is to discover architectural styles in the system under consideration and to deduce which attributes have led to it. Also, for a given set of attributes, analysts must consider which other architectural styles might be appropriate, and to evaluate their advantages and disadvantages compared to the set of styles already used for the system under consideration.

Attributes that are discovered as a result of the design decisions recovery process, and that do not map to the actual needs of the business system, are isolated and emphasized. This is done since architectural styles that traced back to these attributes exist because:

- there is some other architectural influence, such as division of work among developers, or
- these quality attributes have existed before at some point in time, but do not exist any more, and that architecture style is probably a candidate for architectural refactorings.

To record this architectural rationale, we use two documents:

1. a document that relates quality facts through design decisions to the actual solution, which is ideally expressed as a set of architectural styles, and
2. a document that contains alternatives to the existing particular solution.

The first document also includes other discovered, non-attribute, architecture influences and their results in the architecture. Even though this information can be presented in several different formats, we use tables, as they are simplest to build and maintain. Another obvious possibility is to use diagrams, but we do not use them due to the very large amount of information that would add too much noise to the diagram. Also, diagrams are harder to update and maintain.

4.3 Architecture Recovery Process Steps

In order to allow work distribution and separation of concerns, our recovery process is divided into the following steps:

- Abstract architecture recovery, business domain analysis, and run-time aspects analysis — Using requirements engineering techniques, existing documentation, and problem domain expertise, analysts try to produce separately an initial abstract architecture of the system and domain-view lists. One of the main techniques that helps recovery of domain and conceptual architecture is actual use of the system, which aid in creating a high-level run-time architecture. It is important to note that much of the effort invested in early iterations is not on getting these artifacts perfectly right but on providing artifacts that will allow us to perform following steps. After every iteration, set of artifacts produced is refined and improved.

- Concrete and test architecture — Analysts try to map abstract architecture concepts onto source code and to discover mismatches. Responsibilities and invariants source code instrumentation is performed at the same time. These are used to make a summary of dynamic aspects of concrete architecture, and initial diagram of static building blocks is produced. Again, this is done with assumption that these will be refined in following iterations. At the same time, optional test view artifacts are constructed using the same techniques. Iteration is finished with unification of concrete and abstract architecture artifacts to reflect current full understanding of logical architecture of system, and to serve as input artifacts for the following iteration.

- Quality attributes and design decisions recovery — While previous steps were mostly routine observation and analysis, this step depends highly on the analysts' software design and development expertise. Also, it depends on availability of codified architectural patterns and attribute quality knowledge for that particular domain. As a guideline, we will present a minimal set of these as a part of our case study. These two activities are very tightly coupled and the iteration between them is so high that they can be effectively considered as one, unified activity. Artifacts produced are quality facts list related to a set of design decisions, and a list of mismatches that indicate possible external architectural influences.

- Alternative influences and possible design decisions — Using artifacts produced in the previous step and domain-view lists, analysts try to discover possible external architectural influences, and possible design alternatives. The amount of effort invested in this step has to be critically estimated and justified. For example, if no changes to the system are supposed to be made, only very obvious alternatives should be documented. On the other hand, if a need for extensive refactoring is already observed, or if large drifts in the business domain are expected, analysts should try to identify as many external factors and alternative design decisions as possible.

These steps form a recovery iteration, and a whole process consists of several iterations. Each subsequent iteration is more specific and detailed than the previous. A common way is to organize iterations following top-down architectural decomposition of the system. For example, first one is concerned with architecture at the system level, and next one at the level of one of subsystems.

Depending on needs, during this step previously produced artifacts can be presented using different formats for different purposes. For example, if artifacts are supposed to be used for general understanding of the system by new developers, additional diagrams can be created that present information from tables in clearer way. Again, effort invested in creation of additional documentation has to be critically evaluated against cost criteria.

In summary, we have presented the overall organization of our process and high-level artifacts. We have omitted presenting the low-level steps and providing the definitive guidelines on how each step has to be performed due to the paper space limitations. Our goal was to keep the process adaptable, and to stress the importance of agility and use of the different techniques to achieve the same goals. For example, we did not enforce the use of any particular tool and its specific recovery steps for the recovery of the concrete architecture.

5. VALIDATION AND EVALUATION

To evaluate the proposed process, we performed an architecture recovery case study of the X MultiMedia System (XMMS) [32]. The main purpose of this case study was to evaluate our process by applying it to the recovery of a real world system. The main factors that have led to the choice of XMMS as a guinea pig are:

- It is a real, industrial strength, multimedia application in very wide use.
- It is of non-trivial size, 65,000 lines of code, but it is also small enough to be tractable for our purposes.
- It is in active development, and multiple versions of its source code can be obtained.
- It is developed using an open source licence, which means that we can examine it and publish our results freely.

Due to the space limitations, we are not able to include the XMMS architecture recover artifacts in this paper. For all the details about the recovered XMMS architecture, please refer to [29].

5.1 Case Study Validity

In order to improve the validity of the results derived from the conducted case study, we have taken several concerns and issues into consideration when choosing our candidate system:

- The XMMS system was under development and effective use for approximately 5 years. This was important for validation of our process since we were dealing with a relatively stable architecture, which has evolved over time. This implied that we had a possibility to detect features such as obsolete functionality, changes in business goals, and so on.
- The core development team consisted of 3 developers, with contribution from many third parties, and a large feedback from the user community, which is typical among widely used open-source systems. The size of the team was important to insure that the architectural influences did not come only from one source. This conforms to the reality of larger scale development and affects the quality and stability of the architecture.
- There was no architectural documentation. In order to increase the difficulty of the case to which our process was applied to, we chose a system for which there was no architectural documentation. This was done with the assumption that it is harder to perform the architecture recovery of a system for which no architectural information is provided.
- There were many contributions from third parties. We wanted a system that was not developed in isolation, as many of the architectural issues arise when it comes to interoperability, distributed development, and similar. There were many add-ons and pluggins, developed by third parties, which provided many sources of architectural influences and made a more realistic case for the architecture recovery.

5.2 Process Step Issues

The focus of this section is on describing our experience and problems during the execution of each process' activity.

5.2.1 Project Elaboration

As we have discussed at the beginning of our case study [29], we did not perform a detailed analysis of the feasibility of our project and the effort that will be required. Like with all software project management estimations, the precise estimation of the cost of the reverse engineering project is difficult to achieve. Since our process was primarily designed to be used for the recovery of small to mid-sized systems, this estimation is not of the crucial importance — our estimate was that the upper time limit for the recovery of the architecture, for a system of the size similar to the size of XMMS, is one month for a team of three analysts. On the other hand, if our process is to be adapted and used for the recovery of architecture of large systems, one should incorporate cost estimation techniques.

We presented some of the influences on the cost of the recovery at the beginning of our case study [29]. Out of the ones that we mentioned (documentation, quality of code, size of the system, developer's information, and analyst's problem domain knowledge), we found that the analyst's previous knowledge about problem domain and related applications plays a dominant role over the others. Even in the case that all the other influences are positive, it is very hard to recover the architecture if one cannot put the gathered information in the context of the problem domain; and it is very time consuming to learn details about the problem domain.

5.2.2 Analysis of Existing Artifacts

In order to simplify the analysis of the existing documentation, we performed an extraction and classification of the potentially relevant architectural concepts and features. In order to perform this extraction, we needed background knowledge about the properties of functional and non-functional

requirements as they relate to the software architecture. Our main focus was on the analysis of different requirements documents.

Although we had access to and analyzed only a list that summarized the high-level features of the system and user manual, for larger systems one usually has access to other kinds of requirements documents. These include software requirements specification, requirements rationale, data dictionary, etc.

5.2.3 Domain Architecture

When we attempted to extract the architecture of XMMS using the original PBS approach, we did not have any intention to recover the architecture of the problem domain. The problem that we had was that it was not possible to extract the conceptual and concrete architecture without having to search for clues in the problem domain. This resulted in an unorganized exploration of the problem domain concepts, and this exploration was not as efficient as it could be as we were not sure what to look for. Also, we did not make any attempt to preserve this gained problem domain knowledge, which could be reused in many different ways in the future development of the system. Therefore, one of the goals of our new approach was to deal with domain architecture issues.

The first approach to solve this problem focused on only documenting the facts about the problem domain. This did not prove to be an effective approach, as the collected facts were simply listed, without a particular organization, and as such they were not very useful for the recovery of conceptual and concrete architectures. It was difficult to relate one fact to another, and to further relate them to the architectural artifacts. Also, we were not able to distinguish different types of concepts that occur in the problem domain.

In order to solve these problems, we had to introduce a meta-model for our domain architecture, to find a format which will present these fact so that they are directly usable for the recovery of the conceptual and concrete architectures, and to find a systematic way to analyze the domain in the early stages of our process. We solved this problem, and using the proposed steps, successfully recovered and presented the domain architecture of XMMS. We used these recovered domain concepts also to guide the recovery of the other architectural aspects.

One of the drawbacks is that we did not make additional attempts to produce alternative meta-models and formats. This is not to say that our approach is the only one or the ideal one, but it was successful in recovering and organizing these artifacts compared to our initial attempts. There are other possible ways to approach this problem. For example, if we are working on the recovery of the architecture of a system for which there are already existing business domain analysis documents, possibly the most efficient way is to follow the already existing meta-model and artifacts.

5.2.4 Conceptual Architecture

The first attempt for the recovery of the conceptual architecture using the original PBS approach was successful only as far as the static conceptual architecture was concerned. Although not clearly required, we attempted the recovery from the logical perspective even at the early stages of our recovery. This resulted in a static conceptual architecture that did not change drastically during the first iteration of our new approach.

The first disadvantage of the PBS approach is that it did not require any form of the dynamic conceptual architecture recovery. This led to the discovery and presentation of structural components without any emphasis on how they cooperate in order to achieve the overall goals. This lack of analysis of the dynamic aspects of the architecture resulted in a reduced understanding of the underlying reasons of the particular static system decomposition. Our approach tackled this problem of the dynamic architecture using the architectural use cases [29].

The advantage in using the architectural use cases for the presentation of dynamic properties is in the fact that the basic building blocks of these use cases, subsystem responsibilities, are directly derived and produced during the work on the domain architecture artifacts and during the static conceptual architecture recovery. Therefore, the early iterations of our process focused on the subsystem responsibility assignments. On the other hand, later stages focused on the integration of these responsibilities into fully developed architectural use cases.

We found that the particular strengths of the architectural use cases are as an abstract model of the communications that occur within the system and a technique that drives the recovery of the architecture and integrates different architectural aspects. Nevertheless, we found that it is too tedious to document all the use cases manually. Also, we found appealing the use of use case formats other than the one presented in the case study.

Problems that we had during the initial architecture recovery were the integration of conceptual and concrete architectures and the refinement of conceptual architecture based on the results obtained during the recovery of the concrete architecture. Since the original PBS approach was not iterative, the analyst was aiming to recover as good conceptual architecture as possible, followed by the recovery of the concrete architecture. This often resulted in the architectural presentations that are not as focused as they could be and with a large distance between the concrete architecture and the conceptual architecture. In order to deal with these problems, our process encourages the iterative refinement of both kinds of architectures, thus allowing the input of the facts discovered during the concrete architecture recovery into the conceptual architecture. This is in conflict with the PBS approach in which the conceptual architecture is derived only from sources other than source code.

We found that this iterative approach simplifies the recovery process, improves the understanding of particular architectural aspects of the system through the indirect analysis of the dependencies among them, and improves the quality of the presentation. Also, the attempt for bridging the distance among different aspects and unification of views simplifies the understanding of the system, and allow us to present its architecture from different perspectives, *e.g.,* pure concep-

tual logical perspective, pure concrete development perspective, unified logical and development perspectives, etc.

5.2.5 Concrete Architecture

One of the main strengths of the PBS approach is the recovery of the static concrete architecture. As this step of the overall approach is almost completely automated by the PBS tool, our new approach adopted the PBS approach static concrete architecture recovery techniques without significant changes. The main enhancements were introduced because of the need to incorporate and improve systematic dynamic concrete architecture recovery.

As with dynamic conceptual architecture recovery, the PBS approach does not emphasize dynamic concrete architecture recovery. In order to overcome this problem, we introduced the activity of the responsibility assignment to the modules during the recovery process.

The first attempt that we made in order to recover the concrete architecture was focused on the immediate recovery of the facts in a bottom-up fashion. We approached the analysis of the concrete architecture starting directly from the analysis of the source code. This quickly proved to be an inefficient approach because we had difficulty in relating the discovered concepts due to the large amount of recovered information. In order to deal with this problem, we decided to try to approach the problem first in a top-down fashion and then in a bottom-up fashion. This led to the analysis of the source code structure down to the level of modules, and an attempt to relate them to the previously recovered conceptual architecture artifacts. Following that, we used this high-level concrete architecture in order to organize and focus on the recovery of the low-level concrete architecture. This recovery included the analysis of the structure at the function level and the recovery of concrete responsibilities, which were related to the actual functions and modules.

This approach resulted in a static concrete architecture enhanced over the concrete architecture recovered using the PBS approach. This enhancement was in the detailed specification of responsibilities of each module and their relation to the functions contained within each module. Also, during the process, we elevated the abstraction level of the module responsibilities from function level to the module level. This resulted in an improved analysis of the concrete dynamic architecture of the system.

After the extraction of the architectural facts, we used them to navigate the source code, and together with previously documented responsibility information, generate the architectural use cases.

5.2.6 Architecture Rationale

As the primary goal of our process, successful architecture rationale recovery depends directly on the successful recovery of other artifacts. As such, we attempted to recover and present all other architectural artifacts in a form that simplifies and facilitates rationale recovery as much as possible.

Architecture rationale recovery is the most subjective recovery activity performed as a part of our process. While other techniques are mostly of an observational nature, rationale recovery relies upon a large amount of background architectural knowledge. Also, it is one of the components of the process that is very hard to automate in any way. This leads to the situation that successful recovery of rationale depends largely on the capabilities and performance of the individual analysts.

This subjectivity of architecture rationale recovery makes it hard to evaluate. This includes the choice of artifacts, comparison of the quality of produced artifacts obtained using different techniques, performance of different recoveries by different analysts, and so on. Nevertheless, we describe the issues that arose during our recovery of the architecture rationale of XMMS system.

Our rationale recovery approach relies on the iterative analysis of the design decisions that are extracted from the recovered architectural artifacts and the motivations that usually lead to these design decisions. As a major group of these motivations, we have the quality attributes and related theory. We focused on these quality attributes as they are recognized as the major source of architectural decisions.

During the early stages of the rationale recovery, we had a problem with detecting these architectural attributes. This was due to the fact that we were trying to discover the quality attributes directly from the observed design decisions. As the gap between them is large, *i.e.,* a set of quality attributes can be represented using many architecture design decisions, the mapping is often not clear. In order to deal with this problem, we introduced the quality facts, as an intermediate instantiation and a representation of the quality attributes. Although these quality facts simplified the process and improved the efficiency of mapping from architectural decisions to quality attributes, they did not completely remove the ambiguity in some cases. Despite this remaining ambiguity, we were successful in recovery of architecture rationale using this technique, and therefore we did not invest further efforts to find other solutions which would simplify the recovery even more.

6. CONCLUSIONS AND FUTURE WORK

In order to understand and integrate legacy systems into new environments, and to successfully develop applications for rapidly changing business domains using lightweight development processes, we developed and presented a lightweight architecture and evolution recovery process. Our approach was based on attribute theory, as that theory permitted us to systematically tackle and recover the rationale behind architecture and evolution.

The major obstacle during recovery was a lack of automated tool support for architectural use-case recovery. Even though we successfully used this particular technique in order to form a mental model of the dynamic interactions within the system, it required manual navigation through module interdependency references, which was very time consuming. As this technique is very commonly used during actual development, and provides an efficient way for the documentation of high-level interactions within the system, a tool that supports automated use-case recovery should be developed.

Our future work will go in two different directions. The first one is further validation and refinement of our process through its application in different development environments. The second direction is the development of new tools and improvement of existing tool support. Special emphasis will be given to development of a tool for automatic architectural use-case recovery and documentation.

7. ACKNOWLEDGMENTS

We thank Daniel Berry and Anne Pidduck for useful discussions and feedback. Davor Svetinovic was supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC) and Fonds de recherche sur la nature et technologies Québec.

8. REFERENCES

[1] *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, Mar. 2002.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, Boston, Massachusetts, first edition, 1985.

[3] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi. The architecture based design method. *CMU/SEI*, 2000.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, Massachusetts, first edition, 1998.

[5] B. Boehm and H. In. Identifying quality-requirement conflicts. *IEEE Software, Vol. 13, No. 2*, 1996.

[6] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Boston, Massachusetts, second edition, 1994.

[7] F. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, April 1987.

[8] P. Coad and E. Yourdon. *Object Oriented Analysis*. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[9] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[10] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, first edition, 1978.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, Massachusetts, first edition, 1999.

[12] D. Gross and E. Yu. From non-functional requirements to design through patterns. www.hkkk.fi/ mrossi/refsq/f113.pdf.

[13] M. Hammer and J. Champy. *Reengineering the Corporation: a Manifesto for Business Revolution*. Nicholas Brealey P., London, first edition, 1995.

[14] IntelliJ IDEA. http://www.intellij.com/.

[15] R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. *CMU/SEI*, 2000.

[16] M. H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architecture styles. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 225-243*, 1999.

[17] P. Knauber, D. Muthig, K. Schmid, and T. Wide. Applying product line concepts in small and medium-sized companies. *IEEE Software , Vol.17, Iss.5*, pages 88–95, 2000.

[18] P. Kruchten. The 4+1 view model of architecture. www.rational.com/products/whitepapers/350.jsp.

[19] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, N.J., second edition, 2001.

[20] J. Mylopoulos, L. Cheung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of ACM, Vol. 42, No. 1*, 1999.

[21] PBS. swag.uwaterloo.ca/pbs/.

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

[23] Refactoring catalog. www.refactoring.com/catalog/index.html.

[24] Rigi. http://www.rigi.csc.uvic.ca/.

[25] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, Boston, Massachusetts, first edition, 2000.

[26] M. Shaw and G. David. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1996.

[27] SHriMP. http://www.csr.uvic.ca/shrimpviews/.

[28] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Hoboken, N.J., sixth edition, 2001.

[29] D. Svetinovic. Agile architecture recovery. Master's thesis, School of Computer Science, University of Waterloo, 2002.

[30] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architecture repair of open source software. *IWPC*, 2000.

[31] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Vol. 26, No. 10*, 2000.

[32] X MultiMedia system (XMMS). www.xmms.org.

[33] E. Yourdon. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1988.

[34] E. Yourdon and L. Constantine. *Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1979.