Component Diagram Recovery with Dynamic Analysis

Supporting Software Architecture Evaluation

Paul Metselaar

Component Diagram Recovery with Dynamic Analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Paul Metselaar
born in Gouda, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Exact
Molengraaffsingel 33
Delft, the Netherlands
www.exact.com

Component Diagram Recovery with Dynamic Analysis

Author: Paul Metselaar
Student id: 1015494
Email: p.a.metselaar@student.tudelft.nl

Abstract

By evaluating the architecture of a software system, ways to improve the system's quality attributes (such as its performance and modifiability) can be identified and valuable lessons can be learned which may also be applied to other systems. An architecture evaluation requires an up-to-date description of the architecture, which is often unavailable. In such a case, reverse engineering techniques can be used to recover it.

For an effective and efficient recovery and evaluation of an architecture, the scope of the recovery should be narrowed to the parts of the system that are relevant for the evaluation and the recovered architectural views should be useful for a wide range of system stakeholders. This thesis presents a case study, in which these issues are addressed by using dynamic analysis and Prolog to recover architectural views. A survey involving representatives of several groups of stakeholders was conducted to assess the usefulness of a recovered view. The results show that the approach is potentially useful, but that more work is needed to further evaluate it and to make it more usable in practice.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	N. Borota, MSc, Exact International Development B.V.
Committee Member:	Drs. P. R. van Nieuwenhuizen, Faculty EEMCS, TU Delft

Preface

This thesis is the final result of my master's project, carried out at Exact. This project was started up after some discussions with Arie van Deursen, Andy Zaidman, Nenad Borota and Toine Hurkmans, who gave me the opportunity to do my master's project at Exact, on topics in which I was interested: reverse engineering, architecture recovery and architecture evaluation. In short, it has been a great learning experience and has made me even more enthusiastic about these topics.

Like architecture recovery, a master's project (and in particular, writing a thesis) can to some extent be supported with tools, but a significant part of the work has to be done manually. I could not have done this without the help of several people. I wish to thank my supervisors, Andy Zaidman and Nenad Borota. Without their guidance, feedback and patience this project would never have been completed. I also wish to thank the people at Exact, who provided a great working environment. In particular, I wish to thank the people who have participated in the survey. Their feedback was not only interesting, but also encouraging.

Paul Metselaar
Delft, the Netherlands
November 22, 2010

Contents

List of Figures

Chapter 1

Introduction

The architecture of a software system is "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [2]. The architecture has a large influence on the quality attributes of the system, such as its performance, modifiability, security and interoperability. By *evaluating* the architecture of a system, the architectural design decisions that influence these quality attributes can be identified [6]. It is possible to assess if a system based on the architecture has the potential to meet certain requirements concerning its quality attributes, allowing strengths, weaknesses and risks of the architecture to be identified.

Evaluating the architecture of a system in an early stage of its development has several benefits, for instance, fixing a problem early is typically cheaper than fixing a problem late [2, 6]. Evaluation in a late stage, when the system has already been built, can also be useful, for example, to look for new ways to improve the system. The results of such an evaluation could also be applied to other systems. Lessons learned from the evaluation of an old system could be applied right from the start when developing a new system.

To evaluate the architecture of a system, an accurate description of the architecture is needed. Unfortunately, such a description is often unavailable [25]. For example, the available documentation might not describe all parts of the system at the necessary level of abstraction. Sometimes there is no documentation at all. Furthermore, the architectural documentation and the actual source code of the system are typically two separate things, which are maintained independently. If one is changed, the other is not updated automatically to reflect the changes. As a result, they have a tendency to "drift apart" over time [32, 25].

A consequence could be that the results of an architecture evaluation are not valid for the actual system [45]. For example, if the documentation describes a strictly layered architecture, but the actual code violates this strict layering, the outcome of an evaluation might be that the system is highly modifiable, even though it may be hard to modify in practice. In that case, it could also be interesting to determine why the layering was violated. If the overhead incurred by the strict layering prevented the system from meeting new performance requirements, a different approach could be chosen for future systems with similar requirements. This means that, for a meaningful evaluation, it is necessary to recover the architectural documentation that is not available and to verify that the available documentation matches the actual system [6, 45].

1.1 Problem Context

The MSc project discussed in this thesis was carried out at Exact.[1] Exact started serving the entrepreneurial world with information technology in 1984 and has grown from a start-up to a public listed global solutions provider. With employees in 40 countries, Exact serves more than 100,000 customers in over 125 countries and provides solutions in over 40 languages.

Exact develops and maintains several systems. Exact is interested in finding ways to further improve these systems and in learning from experiences with these systems, so that this knowledge can be applied right from the start when building new systems. As mentioned earlier, one way in which this could be supported is by applying architecture evaluation and recovery techniques.

The work discussed in this thesis is a step towards the creation of a repeatable process for the recovery and evaluation of the architecture of existing software systems.[2] The primary goal of the evaluation is to find technical recommendations that can be used to improve the system under analysis or other systems. The evaluation process should focus on evaluating a system's quality attributes, rather than its functionality. Since future systems are likely to have different requirements and a different architecture, the proposed recommendations should be broad, rather than detailed. Architectural documentation that is required for the evaluation must be validated against the implementation or recovered if it is not available.

To identify suitable existing architecture evaluation and recovery processes, the available literature was studied [29]. The conclusions of the literature study form the starting point for this project. The combined recovery and evaluation process can be formed by iteratively applying an architecture recovery process and an evaluation process, so that findings from an architecture evaluation can be used to refine the recovered architectural views in the next iteration, which in turn allows a better evaluation of the architecture [43]. This approach allows a choice to be made from the many evaluation methods that have been published in literature [1, 11, 44], of which the Architecture Tradeoff Analysis Method (ATAM) [2] was found to be the most suitable. A brief overview of the ATAM is given in the next section.

The Symphony architecture recovery process [9] and several reverse engineering techniques that can be applied within the Symphony process were also selected by comparing methods published in literature. Although Symphony is by no means limited to using these techniques, the use of *dynamic analysis* to extract data from the system and the use of pattern recognition techniques to create abstract representations of this data were found to be the most suitable to support an ATAM evaluation, based on their advantages and disadvantages published in literature. These techniques and the Symphony process will be discussed in detail in chapter 2.

In this project, Symphony and the selected techniques, supported by tools, are applied in practice to one of Exact's systems to determine if they can indeed be used to recover and validate architectural documentation that can be used in an ATAM evaluation and if they indeed have the presumed advantages.

[1] `http://www.exact.com`

[2] The term *existing system* refers to any system that has been implemented, regardless of whether it is still maintained. This includes legacy systems, but also systems that have been implemented recently.

1.1.1 ATAM

The Architecture Tradeoff Analyis Method (ATAM) [2] is an architecture evaluation method which supports the evaluation of a wide range of quality attributes of an architecture. In particular, it is designed to identify *tradeoffs* between quality attributes, that is, it can be used to find parts of the architecture where improving one quality attribute could have a negative impact on another. For example, methods to improve modifiability often reduce the performance of the system.

In an ATAM evaluation, the system's quality attribute requirements are specified in the form of *scenarios*. A scenario describes how the system must respond to a particular stimulus in order to meet a particular requirement. Scenarios are somewhat similar to use cases, but where use cases focus on specifying the required functionality of a system, ATAM scenarios focus on specifying non-functional requirements and typically only briefly mention functionality to put the non-functional requirements in context. For example, a performance scenario could specify a maximum response time for a certain action triggered by a user.

Each scenario is analyzed individually. The architect explains how the system would execute the scenario and identifies the parts of the architecture and the architectural decisions, patterns and approaches which are involved (or which would have to be modified to carry out the scenario). At least a cursory assessment is made to determine if they pose any *risk* to the system's ability to meet the requirement specified in the scenario. The actual assessment method is not specified by the ATAM, the analysts are free to choose a technique matching their needs, ranging from informal discussions to in-depth quantitative analysis techniques. Architectural decisions that have a large influence on a quality attribute are identified as *sensitivity points*. If a sensitivity point influences multiple quality attributes, it is identified as a *tradeoff* point. For example, the decision to use an intermediate component to separate two groups of components can be a sensitivity point for modifiability. Because the intermediate component introduces overhead, it can also be a sensitivity point for performance. The decision is a tradeoff point between modifiability and performance, because it influences both quality attributes in opposite ways.

The scenarios used in an ATAM evaluation are generated in two different ways. First, scenarios are derived top-down, from the business goals that motivate the development and maintenance of the system. For example, from the system's business goals it might follow that the system must interoperate with other systems. Scenarios could then specify requirements such as a maximum amount of effort to create an interface with a new system, or a maximum amount of time allowed to exchange a particular kind of data with another system. The scenarios are prioritized and the most important ones are analyzed.

After the first analysis phase, additional scenarios are generated in a brainstorm session involving representatives of all of the system's stakeholders. These scenarios are also prioritized, after which the most important ones are analyzed. Because only the most important scenarios are analyzed, the evaluation is narrowed down to the most important quality attribute requirements and the parts of the architecture which have the largest impact on those quality attributes. This allows the ATAM to analyze large systems efficiently.

Even though it narrows down the scope of the evaluation, the ATAM is a rather heavyweight process [1]. In a process in which multiple ATAM evaluations are performed iter-

atively, this disadvantage could be overcome by initially performing scaled-down versions of the ATAM. The evaluations could then be scaled up in subsequent iterations as a better understanding of the system is obtained. Furthermore, results from previous iterations, such as evaluation scenarios, can be reused in subsequent iterations.

1.2 Research Questions

The main question motivating this master's project is: *How to recover and validate architectural descriptions for use in an ATAM evaluation?* The work discussed in this thesis will address three sub-questions of this main question, in the context of recovering (a part of) the architecture of one of Exact's systems. As mentioned earlier, an architecture recovery process and reverse engineering techniques have already been selected. However, tools to support the process still need to be set up, leading to the first research question.

RQ1 *Which tools can be used to support and (as much as possible) automate architecture recovery using dynamic analysis and pattern matching techniques?*

Answering RQ1 should, in theory, allow the required tool-supported architecture recovery and evaluation process to be set up. Ideally, at least one iteration of this entire combined process would be performed to validate the process, techniques and supporting tools. However, to keep the project within a reasonable timeframe for a master's project, an ATAM evaluation will not actually be performed. This means that another indicator is needed to be able to assess whether the recovery process and tools are likely to be useful to recover documentation for an ATAM evaluation. As discussed earlier, architecture recovery has two main purposes in the context of architecture evaluation: (1) validation of existing documentation and (2) recovery of additional documentation. As a rule of thumb, architectural views which were useful when designing the architecture should also be presented in an ATAM evaluation [2]. If those views can be reconstructed or validated, in a way that is found to be readable and useful by a wide range of stakeholders, it is likely that the views will also be useful in an ATAM evaluation. This way of performing a preliminary evaluation of the recovery process and tools, focusing only on the recovered views, is reflected in the following two research questions:

RQ2 *Can the tool-supported architecture recovery process be used to validate existing architectural documentation?*

RQ3 *Can the tool-supported architecture recovery process be used to recover documentation that is considered readable and useful in practice by the system's stakeholders?*

1.3 Project Objectives

The main objective for this project is to implement a repeatable architecture recovery process which enables validation and recovery of architectural documentation for use in an ATAM evaluation. It should be possible to narrow down the scope of the recovery as much

as possible to allow efficient recovery of architectural views from large systems, while still allowing a thorough evaluation. Ideally, if only a part of the architecture is evaluated, documentation should be recovered (or validated) only for that part, rather than recovering documentation for the entire architecture. The recovered architectural views should be visualized using the Unified Modeling Language (UML) [37], because UML is already used within Exact to document software architectures.

Existing tools should be selected, or new tools should be implemented, to automate the process as much as possible. To validate the recovery process and tools, a case study will be performed, in which they are used to recover architectural views of one of Exact's systems. *As-designed* architectural views and decisions will then be validated against the recovered (*as-built*) view. Ideally, the recovered views should also be immediately usable in an ATAM evaluation involving a wide range of different stakeholders. To determine whether this is the case, a recovered view will be presented to several stakeholders, who will then be asked to rate the extent to which they find the view useful.

1.4 Thesis Outline

This thesis is organized as follows. Chapter 2 will describe the overall architecture recovery process and the recovery techniques applied within this process to recover architectural views for use in an ATAM evaluation. In chapter 3, the design and implementation of the tools used to support the recovery process will be discussed briefly. Chapter 4 presents a case study in which the recovery approach was applied to the Exact Connectivity Layer, a system developed by Exact. The readability and usefulness of one of the views recovered in the case study is evaluated in chapter 5. Chapter 6 discusses related work. Finally, chapter 7 presents conclusions and directions for future work.

Chapter 2

Architecture Recovery Approach

This chapter describes the requirements that must be met by the architecture recovery process for the purpose of supporting an ATAM evaluation. Then, the recovery approach and techniques used in this project will be outlined, with a brief discussion of how they address the requirements.

2.1 Requirements

To be able to support an ATAM evaluation of Exact's systems, the recovery approach must meet the following requirements:

1. *The approach must support the recovery and validation of a wide range of different views.* The ATAM does not prescribe a fixed set of documentation that is needed for an evaluation. [2] recommends several different *views* of the architecture which are found useful in most evaluations, including views describing the static structure of the system and views describing runtime behavior. However, the required documentation can differ between different ATAM evaluations, depending on the goals of the evaluation and the quality attributes and system under analysis.

2. *The recovered views must be expressed in a way that can be understood by a wide range of stakeholders.* Active participation of different groups of stakeholders is essential in an ATAM evaluation [24]. This means that the recovered views must be expressed in terms of concepts with which the stakeholders are familiar. This includes not only the notation with which elements and relations are represented in a view (for which UML must be used), but also their semantics. Although nonconventional views can give new insights into the system [16], the focus is on conventional views to provide a "backbone" for an evaluation and to facilitate validation of existing views.

3. *The approach must help identify the architectural approaches used in the system.* Architectural styles, patterns and tactics typically have well-known influences on quality attributes. This is used as a starting point for analysis in an ATAM evaluation [2].

4. *The process must be repeatable.* Ideally, different people analyzing the same system for the same purpose should obtain the same results. Furthermore, it should be easy to update the recovered views for subsequent versions of the system.

5. *The scope of the recovery must be narrowed down as far as possible.* Since the system under analysis is typically large, recovering the entire architecture is costly. The costs can often be reduced by only recovering the parts of interest for the evaluation.

6. *The recovery process, techniques and tools must be applicable to systems written in VB.NET and ASP.NET.* The systems Exact wishes to analyze are typically written in a combination of those languages. However, these are not the only languages in use at Exact, so it is important to minimize dependencies to a particular implementation language whenever possible.

7. *The recovered architectural views must be accurate.* The views must not contain elements or relations that do not exist in the actual system (false positives). Elements and relations of interest may not be omitted (false negatives). The properties of recovered elements and relations must match the actual implementation.

2.2 Recovery Process

Most architecture recovery methods published in literature describe the recovery of a fixed set of views, based on a fixed set of reverse engineering techniques [9]. To address the first requirement, a more generic architecture recovery process is needed, which can guide the recovery of a wide range of different views, using different techniques if necessary. A process which meets this requirement is the Symphony architecture recovery process proposed by Van Deursen et al. [9]. This process will be used in this project.

Symphony recovers *views* which represent (a part of) the system's architecture. Each view conforms to a *viewpoint*, which specifies the kind of information contained in a view and the rules and conventions used to create, represent and analyze views based on the viewpoint. Symphony distinguishes between source views, target views and hypothetical views. A *source view* can be extracted directly from system artifacts such as source code and execution traces, but it may be too detailed for use as an architectural view. A *target view* describes the as-implemented architecture and contains information needed to solve the problem motivating the recovery effort, at the necessary level of abstraction. A *hypothetical view* is a description of the architecture which is typically obtained by interviewing developers and examining existing documentation. It might be inaccurate, but it can be used to guide the recovery process, or to validate the as-designed architecture or a hypothesis against the as-built architecture represented by a target view.

A set of *mapping rules* specifies how to derive target views from source views. Although the rules can be informal heuristics or guidelines, formal rules are preferred, because they allow the derivation to be performed automatically. When a target view is created, the mapping rules are also instantiated in the form of a *map* which specifies the corresponding facts in the source view for each element and relation in the target view. The views and mappings between them are stored in a *repository*.

2.2.1 Symphony Process

Symphony is an iterative process consisting of two stages: reconstruction design and reconstruction execution. During *reconstruction design* the problem is analyzed and a procedure for reconstructing the architecture is defined. In the *reconstruction execution* stage the architecture is reconstructed by executing the procedure. The two stages are typically iterated, because the results of a reconstruction often reveal new opportunities for reconstruction to be performed in the next iteration.

The reconstruction design stage consists of two activities: problem elicitation and concept determination. During *problem elicitation* the problem motivating the reconstruction is specified. In the first iteration of the iterative architecture recovery and evaluation process, this step could be carried out by performing a scaled-down ("mini") ATAM, focusing on finding a few important scenarios and identifying the kind of documentation needed to analyze them. The mini-ATAM is intended to steer the recovery effort, rather than to actually evaluate the architecture. In subsequent iterations, problem elicitation can be based on findings from previous iterations, for example, if a view is found to be missing certain important information, recovering this information could become the goal of a Symphony iteration.

In the *concept determination* step, the kind of information needed to solve the problem and a way to obtain this information are determined. It consists of five activities:

1. *Identify Potentially Useful Viewpoints.* A list of viewpoints that contain the information needed to solve the problem is made. Viewpoints can be selected from a catalog or be created specifically for the reconstruction. The initial set of viewpoints can be based on suggestions by the stakeholders and the architect. Furthermore, some evaluation techniques which could be applied within the ATAM process, such as a performance model, may require specific information to be present in the target viewpoint.

2. *Define/Refine Target Viewpoint.* The relationships needed in the viewpoints identified in the previous step are listed and prioritized. Duplicates are removed. The most important relationships are incorporated in the target viewpoint.

3. *Define/Refine Source Viewpoint.* The information that is needed in the source views is determined. Since the source views combined with the mapping rules must enable the derivation of the target views, this activity is closely related to the previous and next activity. Furthermore, the source viewpoint depends on which information *can* be extracted (preferably automatically) from the system artifacts.

4. *Define/Refine Mapping Rules.* A set of mapping rules is created, describing how to derive the target views from the source views.

5. *Define Role and Viewpoint of Hypothetical Views.* If a hypothetical view is needed, its role is determined and then its viewpoint is defined.

The explicit creation of a reconstruction procedure allows it to be reused, for example, in subsequent iterations, in the analysis of different versions of the system or the analysis of similar systems [9]. This aspect of Symphony directly addresses requirement 4.

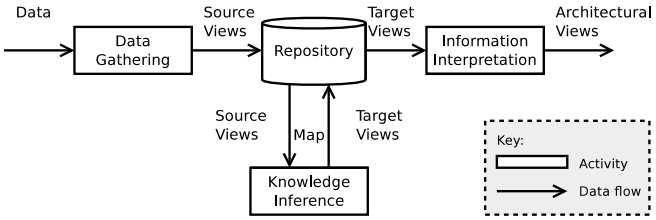


Figure 2.1: Symphony reconstruction execution data flow [9]

In the reconstruction execution stage the actual reconstruction is carried out. The individual activities in this stage and the data flow between them are shown in figure 2.1. The source views are extracted from the system and stored in the repository (*data gathering*). The target views can then be populated by applying the mapping rules to the source views (*knowledge inference*). While creating the target views, a map between the source and target views can be created. For example, the map could list each class in the source view and the layer in the target view to which the class belongs. The map is typically created iteratively. Each iteration refines the map or raises its level of abstraction until the target view can be created. Finally, the target views are presented in a way that allows the problem motivating the reconstruction to be solved (*information interpretation*).

[9] lists a wide range of techniques that can be used to perform the activities in the Symphony process, but does not prescribe the use of any particular technique. For example, any data gathering technique that delivers source views conforming to the source viewpoint can be used. This makes Symphony a very flexible process, addressing requirement 1.

The reverse engineering techniques used in this project will be discussed in the following sections. These techniques were selected by comparing existing techniques published in literature [29]. Although they should in theory support the recovery of several different kinds of views (requirement 1), it is certainly possible that other techniques are more useful for recovering specific views necessary for a particular evaluation. In practice, the necessary techniques and tools will typically be selected during the reconstruction design phase of Symphony, possibly based on experiences with other techniques in an earlier iteration.

2.3 Data Gathering

Data gathering techniques include *static analysis*, which involves the analysis of the system's artifacts such as its source code, and *dynamic analysis*, where data is collected about the system as it executes [9].

To perform dynamic analysis, a system is instrumented, allowing data of interest to be logged while a particular execution scenario is performed with the system. The execution scenarios could be derived from the scenarios written in the (scaled-down) ATAM evaluation. A common dynamic analysis approach is *tracing*, which involves logging the method calls made by the running system, resulting in an *execution trace*. It is also possible to keep track of other things, such as the amount of memory used and the amount of time spent

in each method, as is commonly done by *profilers*. Presenting such performance data in terms of architectural concepts [46] could be useful if performance is found to be an issue in an ATAM evaluation, but that is beyond the scope of this project. The focus is on tracing method calls and object creation, so that the system's structure and behavior can be studied. This will be discussed in more detail in section 3.1.

Cornelissen et al. mention two advantages of dynamic analysis [7]. First, dynamic analysis can give precise results when a system makes use of polymorphism, which is common in object-oriented systems, including Exact's systems. For example, when an interface is implemented by several classes and one of the methods of the interface is called, it is often hard to determine statically which implementation will be called. This can be observed more easily at runtime.

Second, it supports a *goal-oriented strategy*, in which only the relevant parts of a system are analyzed. In an ATAM evaluation, we may only be interested in the parts of a system that are involved in a particular scenario. Dynamic analysis facilitates narrowing down the scope of the recovery to those parts, because only the methods that have actually been called while performing the scenario are included in a trace. This addresses requirement 5.

Four disadvantages are also mentioned [7]. First, dynamic analysis tends to be incomplete, because there are no guarantees that the results obtained for a particular set of scenarios are valid for all executions of the system. As a result, important system behavior could be missed in an evaluation. Furthermore, it may be hard to answer more general questions about the system that are not specific to a particular scenario, but which may still surface during an ATAM evaluation.

Second, it may be difficult to establish a set of scenarios that trigger the parts of the system of interest. If not all relevant parts are properly triggered, the eventual results of an evaluation may be inaccurate. If too many parts are triggered, the benefit of a goal-oriented strategy is lost. Deriving execution scenarios from ATAM evaluation scenarios and determining whether a set of execution scenarios sufficiently covers a set of ATAM evaluation scenarios may not be straightforward. Examples include modifiability scenarios which prescribe a maximum amount of effort to make a particular change to the system, or *exploratory* scenarios which represent radically new requirements.

Third, dynamic analysis can involve large amounts of data, causing scalability problems. As will be discussed in chapter 3, filters are applied to reduce the number of method calls that have to be processed. Furthermore, rather than presenting the raw trace data to the stakeholders, more abstract representations are presented, as discussed in the next section.

Finally, the results obtained with dynamic analysis may be influenced by the *observer effect*, where a system behaves differently when under observation. For example, the overhead caused by the instrumentation code may result in timeouts which would otherwise not have occurred.

Because the goal-driven approach enabled with dynamic analysis is potentially a large benefit in the context of an ATAM evaluation, this project will focus on dynamic analysis. As will be discussed later, static analysis could also have been useful in the case study discussed in chapter 4. Because a combination of static analysis and dynamic analysis is not feasible in this project, it is left as future work.

2.4 Knowledge Inference

Architectural constructs are typically not directly expressed in source code (for example, programming languages typically do not have an explicit "layer" construct), therefore, techniques are needed to raise the level of abstraction from low-level facts extracted from the system to high-level architectural specifications [25]. Ideally, a knowledge inference technique would be able to fully automatically recover any desired architectural view, given only the facts extracted from the system. Unfortunately, no such "silver bullet" technique is known at the moment.

Abstraction techniques range from (almost fully) manual techniques to (almost fully) automatic techniques [9, 39]. Drawbacks of manual techniques include the typically large amount of work involved in recovering the architecture of large systems, the limited (if any) support offered for the identification of architectural approaches (requirement 3) and typically poor repeatability (requirement 4). With automatic techniques, such as the use of clustering algorithms to create subsystem decompositions, there is a risk that the recovered subsystems are not meaningful to a human analyst (requirement 2) [3]. This limits the use of the recovered views for the purposes of architecture evaluation and the validation of existing architectural views.

In between are several semi-automatic approaches, which typically involve manually writing specifications of how instances of architectural elements, relations and patterns of interest can be recognized automatically from the facts extracted from the system. For example, *rules* or *queries* can be written to match elements and relations of a particular architectural *style* [20, 45]. In the case of the client-server style, rules could be written to look for patterns of method calls which are involved in establishing a connection between a client and a server. Such pattern matching approaches can be applied in several ways:

- An existing library of rules can be used to "discover" which architectural concepts are present in the system [20], for example, to get an initial idea of the architecture of a system if very little is known about it. By using a set of rules which support recognition of elements and relations in a particular architectural style, it is possible to determine if a system implements a particular style and if so, to determine which parts of the system are responsible for implementing it [20].

- If some knowledge is available about the architectural styles used in the system, a specific ruleset can be selected or developed to determine how the system's implementation maps to the concepts of a particular style [20]. Furthermore, if a particular architectural element has been identified manually, rules can be written to automatically find other instances of the element in the system.

- Contrary to "static" documentation, new views can be generated from extracted facts as needed to answer questions about the system [20]. This can be done by using a library of recognition rules as discussed earlier, or by creating a system-specific ruleset. In the latter case, the rules themselves are a form of documentation, formally specifying the way particular architectural concepts have been implemented in the system.

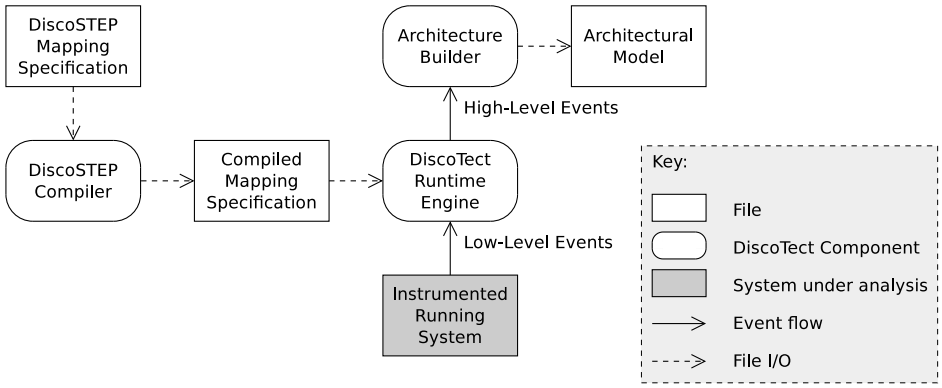


Figure 2.2: Discotect data flow diagram [45].

Since the abstractions to be recognized are defined manually, it is possible to recover architectural views in terms of concepts with which the system's stakeholders are familiar (requirement 2). There is also limited support for the recognition of architectural approaches (requirement 3), as long as rules are available which can recognize the approaches used in the system. Furthermore, once recognition rules have been written for a particular system, they can be reused in the analysis of similar systems or newer versions of the same system (requirement 4).

Pattern matching approaches also have disadvantages. First, there are typically many ways to implement an architectural concept, even within a single implementation language. If the rules do not sufficiently cover the possible implementations, the architectural concepts might not be recognized (false negatives). If the ruleset is too generic, there is a risk of recognizing instances of concepts which do not actually exist (false positives). Assessing whether a ruleset achieves sufficient coverage is, in general, not easy. Furthermore, if a system deviates from the "standard" version of an architectural concept, it may also be missed. However, this can sometimes be applied usefully if the goal is to verify whether certain architectural concepts have been implemented correctly.

Second, there appears to be no publicly available set of rules that can recognize a substantial number of architectural styles in .NET software. Unless a ruleset can be reused from the analysis of a similar system, a ruleset will have to be developed from scratch, resulting in a significant start-up cost.

2.4.1 DiscoTect

Schmerl et al. propose a rule-based technique and tool (called DiscoTect[1]) which can be used to generate architectural descriptions based on observed runtime events [45]. Figure 2.2 gives an overview of the approach. DiscoTect applies a set of *mapping rules* to transform low-level events (such as method calls occurring in the system under analysis) into events that are meaningful at the architectural level (such as establishing a connection

[1] `http://able.fluid.cs.cmu.edu:8080/Able/DiscoTect`

between a client and a server). The low-level events are obtained with dynamic analysis, as discussed in section 2.3. The high-level events generated by DiscoTect are sent to an *architecture builder*, which uses them to incrementally construct an architectural view of the system under analysis. The language used to specify mappings between low-level and architectural events is discussed in more detail in section 3.3.

2.5 Information Interpretation

Schmerl et al. [45] propose the use of DiscoTect to recover component-and-connector views, which describe a system in terms of high-level runtime elements (such as clients, servers, repositories, processes, etc...) and the interactions between them (such as communication using a particular protocol, retrieval and storage of data, synchronization, etc...). They represent the recovered views in the Acme architecture description language (ADL), using the AcmeStudio tool as an architecture builder and to visualize the views.

Within Exact, UML is typically to represent architectural views. This means that recovering views in UML, rather than Acme, will make it easier to validate existing architectural views. Furthermore, an architecture evaluation is likely to benefit from presenting architectural views in a notation with which the system's stakeholders are familiar (requirement 2). Fortunately, DiscoTect is not limited to generating Acme models, but in order to generate UML models, a different architecture builder and visualization tool will have to be used. One of the UML modeling tools used within Exact is Enterprise Architect[2] (EA). This tool supports importing UML models in XMI (XML Metadata Interchange) format [35]. Section 3.2 will describe how UML models in XMI format can be generated with DiscoTect, so that Enterprise Architect can be used to visualize the generated models.

[2] `http://www.sparxsystems.com`

Chapter 3

Architecture Recovery Tools

Many of the architecture recovery techniques described in the previous chapter can to some extent be supported with tools. This chapter discusses a set of supporting tools and how they interoperate. Chapter 4 presents a case study in which the Symphony process, supported by these tools, is used to recover a part of the architecture of a system developed by Exact.

3.1 Execution Tracer

Section 2.3 discussed the use of execution tracing to extract data from a system. Several techniques for obtaining execution traces have been published in literature [53] and many of them can be used in a .NET environment [29]. Techniques such as Aspect-Oriented Programming (AOP) can be used to automatically insert (*weave*) tracing code into existing code. By creating an aspect which adds logging code at the beginning and end of every method, each time a method is entered or left can be logged. Some AOP frameworks also support adding code around method calls, which allows tracing calls to methods without modifying them. This approach is useful when it is easier to modify the caller than the callee, for example when tracing calls to methods in the .NET Framework Class Library.

Another option is to use the Profiling API [31] provided by the .NET CLR (Common Language Runtime, the virtual machine in which .NET applications run), which allows monitoring events that occur at runtime. The profiling API can be used in two different ways to obtain an execution trace. The first approach is to register callbacks that are called whenever a method is entered or left. The callbacks simply log the events. In the second approach, a callback is registered that adds tracing code to each method just before it is JIT-compiled [38]. It is similar to the AOP-based approach mentioned earlier, except that weaving takes place at runtime. Both approaches can be used to trace the execution of a system without modifying its (possibly signed) assemblies. This includes tracing code inside the Framework Class Library and code generated at runtime. However, using the Profiling API typically introduces more overhead than an AOP-based approach, because the system is instrumented at runtime. The additional overhead could cause problems such as timeouts that would otherwise not have occurred (also known as *observer effects* [7]).

Several existing tracing tools have been examined in [29]. While there is no shortage

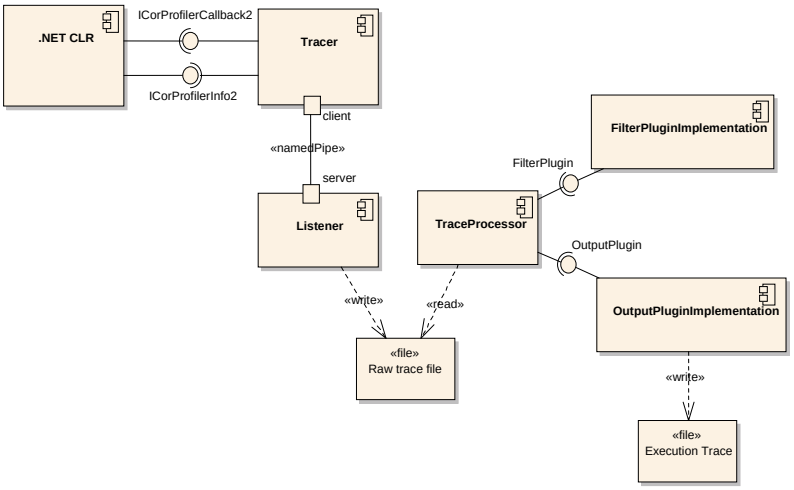


Figure 3.1: Execution Tracer (UML Component Diagram).

of profilers which provide aggregated information such as the amount of time spent in a particular method, only a few tools provide a "raw" trace of method calls. Unfortunately, practical problems were encountered with all of the tracing tools that were found. Therefore, the decision was made to develop a simple tracer.

Because a tracer based on the Profiling API can give very complete traces in a straightforward way, including calls to and from methods in the Framework Class Library, without having to modify any (possibly signed) assemblies, this approach was chosen. Since registering callbacks for method enter/leave events directly gives access to the runtime events of interest, this approach appeared to be the most straightforward way to implement a tracer.

Figure 3.1 gives an overview of the execution tracing environment. The actual execution tracer is based on the profiling architecture outlined in the documentation of the Profiling API [31]. It consists of two components: the Tracer and the Listener. The Tracer component uses the Profiling API to monitor events occurring in the running system and sends the events to the Listener component through a named pipe. The Listener component writes the events received from the Tracer to a file (using a simple binary format). It also provides a GUI which allows the user to control the Tracer. The TraceProcessor component reads the trace file written by the Listener, optionally applies filters to it and writes it in a format that can be used by other tools, such as DiscoTect. It allows different filters to be applied to the same trace and allows the same trace to be converted into different formats. The actual filters and output formats are implemented in separate plugins.

The individual components are described in more detail in the remainder of this section.

3.1.1 Tracer

The Tracer component is responsible for communicating with the .NET CLR. Running in the same process as the application that is being traced, it receives notifications of runtime events from the CLR, optionally applies some filtering to them and sends the events which

pass the filter to the Listener. As prescribed by the Profiling API documentation [31], the Tracer only gathers data, but (besides filtering) does not analyze the trace.

Communication with the CLR is done using two APIs. The CLR profiling API [31] is used to monitor runtime events. It allows a COM component[1] which implements the `ICorProfilerCallback2` interface[2] (in this case, the Tracer component) to register callbacks which are called by the CLR whenever certain runtime events occur.

On initialization, the Tracer is given a pointer to an instance of an implementation of the `ICorProfilerInfo2` interface. This interface can be used to obtain information about a runtime event, such as the name of the method that was called.[3] Not all relevant information is provided through the `ICorProfilerInfo2` interface. The Metadata API [30] is used to obtain additional information, such as the types of the parameters of a method.

Below, a brief description is given of the kinds of runtime events monitored by the Tracer and the way the events are handled.

Module and Class Loads The Profiling API supports callbacks which are called by the CLR whenever a new module or class is loaded. In these callbacks, the Tracer retrieves information about the thing that has been loaded (such as the ID and name of a class) and sends it to the Listener. This way, the Tracer only has to extract and send this information once, rather than every time a method of the class is called.

Function Information A similar approach is used to extract and send information about functions. Although an attempt is made to extract as much information as early as possible, some information about generic methods can only be extracted when the method is actually called. If a method is part of a generic class, or the method itself has type parameters, multiple instances of the method may not only have the same IL code, but after JIT compiling they may also share the same native code and have the same FunctionID. For example, if `List`, `Customer` and `Order` are classes, the methods `List<Customer>.Add` and `List<Order>.Add` will have the same FunctionID. The Profiling API does not provide an ID which uniquely identifies a method in this case. However, the information needed to distinguish between methods with the same FunctionID is passed to the function enter callback. This callback assigns a unique ID to each method and makes sure that information about the method, such as the ClassID of the method containing the class, is only sent to the Listener once.

The Metadata API provides an easy interface for enumerating the parameters of a function and extracting their names. However, the Profiling and Metadata APIs do not provide a similar interface to extract the types of parameters. The Metadata API does provide access to a binary representation of the signature of the function, which includes the types of the parameters of the function. The format of the binary signature is defined in ECMA-335,

[1] `http://msdn.microsoft.com/en-us/library/ms680573.aspx`

[2] The interface that must be implemented depends on the version of the .NET framework targeted by the profiler. Exact uses .NET 3.5SP1, which requires `ICorProfilerCallback2`.

[3] Unfortunately, the reflection functionality provided by the Framework Class Library cannot be used, because the event callbacks may not be written in (or call) managed code.

Partition II [13]. The Tracer parses the binary signature so that the function enter/leave callbacks can correctly extract the values of the parameters (and the return value). An advantage of the availability of the binary signature is that there is no need to invent a new format to send parameter types to the Listener. The binary signature is sent to the Listener with only a few trivial modifications. Discussion of these modifications and how the signature is parsed by the Tracer are beyond the scope of this thesis.

Object Creation and Garbage Collection Each object created at runtime is identified by an ObjectID, which is simply a pointer to the object's location in memory. During garbage collection, the CLR may move live objects in memory, changing their ObjectIDs. Having to deal with changing IDs complicates trace analysis, therefore the TraceProcessor will assign an ID to each object that remains constant throughout the trace. To allow keeping track of objects as they are moved in memory, the Profiling API supports a `MovedReferences` callback, which notifies the Tracer whenever a block of memory is moved. This information is then passed to the Listener.

Thread Creation/Destruction In order to support multithreaded code, the Tracer monitors thread creation and destruction. A separate call stack is kept for each thread.

Method Enter/Leave Whenever a method is called, the CLR calls the `FunctionEnter2` callback. This callback first determines which method was called and pushes an object representing the method call on the call stack of the current thread. Then, it checks whether the call matches certain filtering criteria. If this is the case, the values of the parameters passed to the method are extracted[4] and a method call event, including the extracted information, is sent to the Listener.

Filtering is often necessary, because extracting information for all method enter events and sending it to the listener can cause a prohibitively large amount of overhead. Different filters are implemented in separate classes, facilitating the addition of new filters.

When a method is left, the method call object is popped off the thread's call stack. A method leave event is sent to the Listener only if the method call matched the filter. If the method returns a value or throws an exception, it is extracted and sent to the Listener.

Communication with the Listener

Communication with the Listener is done via a named pipe. The Listener acts as the named pipe server. When a Tracer instance is initialized, it connects to the Listener. Multiple Tracer instances may be active at the same time.

A simple binary protocol is used to send events from the Tracer to the Listener. Each event consists of a single byte, which indicates the type of event, followed by event-specific data.

The Listener can send commands to the active Tracer(s) using a similar protocol. At the moment, the only command that has been implemented allows the user to insert a marker

[4] At the moment, only primitive types, strings and object IDs are extracted. Member fields of classes and value types are not extracted yet.

into the trace. For example, a marker could be inserted between each action performed in a scenario. There are two reasons why markers are not inserted into the trace file(s) by the Listener itself. First, it would require the Listener to "understand" the binary trace format. Implementing this would be a large amount of work. Second, the Tracer uses buffered IO when sending events. It only sends events to the Listener once its buffer is full. As a result, the Listener cannot reliably determine where to insert the marker. This is also the reason why the Tracer and Listener do not use buffered IO for sending and receiving commands.

More commands could be added in the future, for example to change the filter at runtime.

Discussion

Implementing a tracer based on enter/leave callbacks was indeed fairly straightforward. However, extracting the types and values of method parameters required more work than expected because the Profiling and Metadata APIs only provide low-level access to this data. Parameter extraction was not fully implemented, but this did not cause significant problems during the case study (discussed in chapter 4). Parameter extraction might be easier to implement when injecting tracing IL code at the beginning and end of each method at runtime. The tracing code could then be implemented as managed code, which can obtain parameter types and values without having to deal with the details of the way data is stored internally by the CLR. However, it is not clear whether this would outweigh the additional effort needed to implement IL injection. Analyzing differences in runtime performance of the two approaches would be interesting as well.

IL injection could also be used to trace access to member variables of classes, which is not possible using enter/leave callbacks. This was not found to be a problem during the case study, but it is possible that this information is important when analyzing a different system, or when performing a different kind of analysis.

3.1.2 Listener

The Listener writes event data received from the Tracer(s) to file(s). It acts as a named pipe server, to which tracers can connect and send runtime events. Multiple Tracer instances can connect to the Listener at the same time, so that multiple programs can be traced concurrently. The traces sent by the different Tracers are written to separate files. The Listener does not process the traces in any way, this is done by the TraceProcessor.

The Listener provides a GUI (shown in figure 3.2) with which the user can control the Tracer(s). It allows the user to specify the output path for trace files, register and unregister the Tracer COM component, start and stop listening and insert markers into the trace(s). Markers are sent to all Tracers which are connected to the Listener.

The GUI can be used to start a program with a Tracer enabled. This is done by setting two environment variables prior to starting the program. These environment variables[5] tell the CLR to activate profiling and to use the Tracer component as a profiler.[6] It is also

[5] `http://msdn.microsoft.com/en-us/library/bb384689%28v=VS.90%29.aspx`

[6] To avoid an infinite recursion, these environment variables are cleared prior to starting the Listener itself.

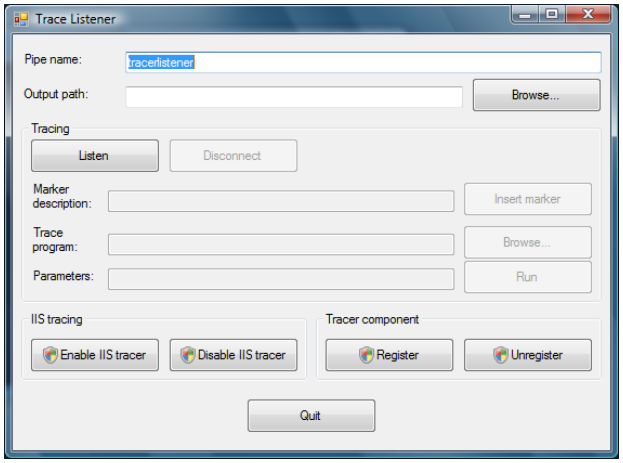


Figure 3.2: Screenshot of the Listener.

possible to enable or disable tracing of code running under an IIS server running on the same machine as the Listener. In that case, the Listener will set (or remove) the necessary registry values[7] and restart the IIS server. This allows tracing of all managed code running under the IIS server, including ASP.NET code.

3.1.3 TraceProcessor

The TraceProcessor converts a binary trace file into a format that can be read by trace analysis tools such as DiscoTect. The TraceProcessor supports different output formats and filters, allowing the same trace to be analyzed with different tools, possibly focusing on different parts of the trace. This avoids having to re-run the entire scenario multiple times. Re-running a scenario not only has the disadvantage of taking more time, the traces could also be slightly different, for example if the program under analysis is multithreaded. Therefore, the filtering implemented in the Tracer is only intended to reduce tracing overhead to an acceptable level. Further filtering is done with the TraceProcessor.

As is shown in figure 3.1, the TraceProcessor consists of a main component which uses plugins which implement different filtering criteria and output formats. The main TraceProcessor component is responsible for reading the binary trace file and passing the events to the plugins in such a way that the plugins can operate independently and do not need to be concerned with the details of the binary trace format or the Profiling API.

Figure 3.3 shows a partial UML class diagram of the TraceProcessor. The classes in the `Events` namespace hide the details of the binary trace format from the rest of the TraceProcessor. Each time an event is read, the `EventReader` reads the event type and creates an instance of the corresponding event class. The constructor of the event class will then read the event-specific data.

[7] `http://social.msdn.microsoft.com/forums/en-US/vsdebug/thread/c5726c6a-0b03-4b50-abc2-c98ed07e3eaf/`

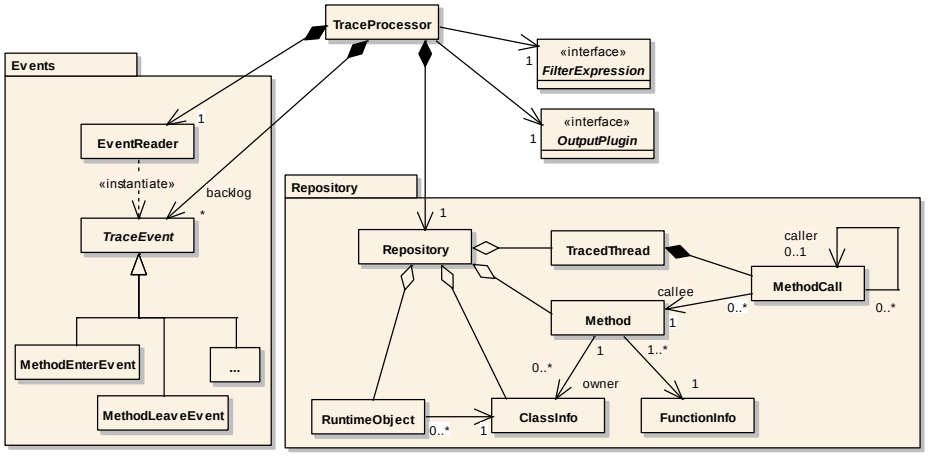


Figure 3.3: TraceProcessor class diagram.

The classes in the `Repository` namespace represent runtime entities of the traced program, such as methods, objects, threads and method calls. Instances of these classes are created by the event parsing code and passed to the plugins, allowing them to retrieve information such as method signatures without having to deal with the details of how this information is represented by the CLR. The `Repository` class keeps track of the known runtime entities while processing the trace, allowing the event parsing code to add, retrieve and remove items to/from the repository, referring to them using the IDs found in the trace file. Because entities can be removed from the repository while processing the trace (for example, when a thread is destroyed), the plugins need to keep track of the entities passed to them if they need to access them later, rather than looking them up in the repository.

The `TraceProcessor` class is in control of the overall process. It uses the `EventReader` to read the events from the trace file. After reading an event, it calls the `fire` method of the event object to process the event. For example, a `ClassLoadEvent` will add a class to the `Repository`. A `MethodEnterEvent` will notify the `TracedThread` object representing the thread on which the call occurred, which will in turn pass the method call to the filter plugin and, if the call matches the filter criteria, pass it to the output plugin.

The Profiling API sometimes calls event callbacks in an unexpected order.[8] This can complicate the processing of the trace file because, for example, an event referring to a particular ClassID may be encountered prior to the `ClassLoaded` event which contains information about the class with that ClassID. To make sure that plugins do not have to deal with this, the `fire` method of an event first checks whether all IDs it needs are present in the `Repository`. If an ID is unknown, the `fire` method will fail without altering the `Repository` or notifying the plugins. The `TraceProcessor` will then add the event to a *backlog* of events that could not be fired and move on to the next event in the trace file. Each time an event is successfully fired, the TraceProcessor will attempt to fire the events left in the backlog. Events remain in the backlog until they can be fired successfully. Although

[8] `http://msdn.microsoft.com/en-us/library/bb397913%28v=VS.90%29.aspx`

this is certainly not the best approach (in terms of performance and correctness), it is quite simple and it was not found to be problematic for the traces analyzed in the case study.

Plugins Filter plugins must implement methods that are called when a *thread creation*, *thread destruction*, *marker* or *method enter* event is successfully fired. Of these events, only method calls can be filtered. However, the plugin can keep track of the other events it receives, for example to only pass calls that occurred after a particular marker was encountered. Simple filter expressions can be created in which multiple filter plugins can be combined, for example to find all calls to methods of classes in namespaces starting with the name `Exact`, that occurred after a particular marker in the trace.

If a call to a method is excluded by the filter, calls made from that method can still be included. For example, if the call to a method `foo` is included, but the call from `foo` to `bar` is excluded, then all included calls from `bar` will be represented as calls from `foo`.

Output plugins must implement methods for the same set of events and in addition must implement methods that are called when a method is left (due to a normal return, tailcall or exception). It receives a notification of all events, except for method enter and leave events, which are only passed to the output plugin if the method call matched the filter criteria. A description of the implemented output formats can be found in appendix B.

3.2 Architecture Builder

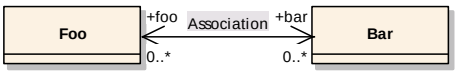
The previous section discussed the observation and logging of *runtime events* which occur when the system under analysis is running. Tools such as DiscoTect "translate" these low-level runtime events into high-level *architectural events* that describe operations on architectural concepts. An *architecture builder* [45] is then used to (1) incrementally create a representation of the system's architecture based on the high-level events and (2) to visualize the recovered architecture.

Schmerl et al. [45] use the AcmeStudio tool for both tasks, resulting in architectural descriptions in the Acme ADL. They represent architectural events in XML format. For example, when a connector between two components is recognized, a `<create_connector>` element is generated, containing information about the name and type of the connector to create and the identifiers of the components to which to attach the connector.

The remainder of this section describes an architecture builder which constructs UML models. The models are stored in XMI (XML Metadata Interchange) 2.1 format [35] and then imported into Enterprise Architect, which can visualize several UML 2.1 diagram types, such as class diagrams, component diagrams and sequence diagrams.

3.2.1 Generating XMI

Several interrelated standards are involved in the representation of UML models in XMI format. Every UML model is an instance of the UML metamodel, defined in the the UML Superstructure [37]. For instance, the UML metamodel specifies which elements and relations can be used in UML models, such as classes, components and associations. The UML Superstructure itself is modeled using the UML Infrastructure [36], which can be seen as



(a) Example class diagram

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
         xmlns:uml="http://schema.omg.org/spec/UML/2.1">
    <uml:Model xmi:id="model_example" name="example" visibility="vis_public">
        <packagedElement xmi:type="uml:Class" xmi:id="class_Foo" name="Foo" visibility="vis_public">
            <ownedAttribute xmi:type="uml:Property" xmi:id="class_Foo_bar" name="bar" type="class_Bar"
                            visibility="vis_public" association="association_Association">
                <upperValue xmi:id="class_Foo_bar_uv" xmi:type="uml:LiteralUnlimitedNatural" value="*" />
                <lowerValue xmi:id="class_Foo_bar_lv" xmi:type="uml:LiteralInteger" value="0" />
            </ownedAttribute>
        </packagedElement>
        <packagedElement xmi:type="uml:Class" xmi:id="class_Bar" name="Bar" visibility="vis_public">
            <ownedAttribute xmi:type="uml:Property" xmi:id="class_Bar_foo" name="foo" type="class_Foo"
                            visibility="vis_public" association="association_Association">
                <upperValue xmi:id="class_Bar_foo_uv" xmi:type="uml:LiteralUnlimitedNatural" value="*" />
                <lowerValue xmi:id="class_Bar_foo_lv" xmi:type="uml:LiteralInteger" value="0" />
            </ownedAttribute>
        </packagedElement>
        <packagedElement xmi:type="uml:Association" xmi:id="association_Association" name="Association"
                         memberEnd="class_Foo_bar class_Bar_foo" />
    </uml:Model>
</xmi:XMI>
```

(b) XMI representation

Figure 3.4: Example class diagram and its XMI representation.

a meta-metamodel: a modeling language which can be used to model metamodels. It is reused in the Meta Object Facility (MOF) [34],[9] so that the UML Superstructure, UML Infrastructure and MOF itself can be seen as instances of MOF, or in other words, they are *MOF-compliant* metamodels. The MOF/XMI mapping [35] specifies how instances of any MOF-compliant metamodel (in this case, UML models) can be written (*serialized*) in a format based on XML.

It is beyond the scope of this thesis to give a detailed description of how to serialize UML models in XMI format. Instead, a simple example will be given to illustrate some of the problems involved in generating XMI with tools such as DiscoTect and how these problems can be solved or worked around.

Figure 3.4 shows a simple class diagram, consisting of 2 classes and an N:M association between them that is navigable in both directions, along with an XMI representation of the diagram. Unfortunately, even though DiscoTect generates XML, it cannot be used to directly generate such an XMI representation. The reason for this is that DiscoTect is designed to generate XML representations of events. Once a piece of XML has been generated, it is sent to the architecture builder and cannot be modified anymore. If the analysis were to start out by creating an empty `<uml:Model />` element, it would be impossible to later add child elements to it, when classes or associations are recognized. Therefore, rather than generating the entire XMI document directly, it is necessary to generate events and use a separate architecture builder to create the final XMI document, similar to what is done by Schmerl et al. when generating Acme models [45].

[9] MOF 2.0 uses UML Infrastructure 2.0, not 2.1.

```
<uml:Model xmi:id="model_example" name="example" visibility="vis_public" />

<xmi:difference xsi:type="xmi:Add" addition="class_Foo" target="model_example" position="-1" />
<xmi:difference xsi:type="xmi:Delete" target="dummy_Package" />
<uml:Package xmi:id="dummy_Package" name="Dummy" visibility="vis_public">
    <packagedElement xmi:type="uml:Class" xmi:id="class_Foo" name="Foo" visibility="vis_public" />
</uml:Package>
```

Figure 3.5: Example XMI addition.

Architectural events can be represented with XMI *differences*, which can be used to specify changes to a model in terms of addition, replacement or removal of elements. Figure 3.5 shows an example, in which a class Foo is added to an empty model.[10] Each of these XML elements can be created by DiscoTect. A separate tool can add them as children to an `XMI` root element and add a proper XML declaration to create a valid XMI document which can be read by tools which support XMI differences.

Unfortunately, XMI differences are an optional part of the XMI standard [35] and Enterprise Architect does not support them. A simple tool (called XMIMerge) was developed to process the XMI additions and deletions, producing a final XMI document that can be imported by Enterprise Architect. It simply finds all `xmi:Difference` elements. If it is a delete operation, it finds the element which has an `xmi:id` attribute equal to the `target` attribute of the `xmi:Difference` element and removes it. In case of an addition, the element referred to by the `addition` attribute of the `xmi:Difference` is moved to the specified location as a child of the target element. In both cases, the `xmi:Difference` element is removed.

The tool lacks support for some XMI features, including replacement operations, but this was not found to be problematic in the case study discussed in chapter 4.

3.2.2 Diagram Layout

While the DiscoTect and Prolog rulesets discussed in the following sections do recover UML models, they do not automatically generate diagrams with a clear layout. Manually adjusting the layout tends to get tedious, particularly in an iterative architecture recovery process where a model is often based on a model created in a previous iteration, so almost the same layout work has to be done over and over again. Fortunately, it is also possible to take advantage of the similarity of the models by automatically copying over the positions and sizes of model elements that have not changed since the previous iteration.

The XMI standard does not specify how to store the layout of UML diagrams. For instance, the fact that class `Foo` is positioned to the left of class `Bar` in the class diagram in figure 3.4 cannot be expressed in its XMI representation. The Diagram Interchange (DI) standard [33] specifies how to store diagram layout, but this standard is not supported by Enterprise Architect. Instead, Enterprise Architect uses tool-specific extensions to store the diagram layout in XMI documents.

[10] The example on page 39 of [35] adds a `uml:Class`, rather than a `packagedElement` element, which is probably an error in the XMI standard: `http://www.omg.org/issues/mof2xmi-rtf.open.html#Issue9690`

The CopyLayout tool was developed to automatically copy the layout from one model to another. It takes two XMI files as input: a *template model* which must contain layout information in Enterprise Architect's format and an *input model* to which to copy the layout information. CopyLayout simply iterates through each model element in the input model and if an element of the same type and with the same name exists in the template model, it copies the element's position and size to the input model.[11]

3.3 DiscoTect

The previous sections have explained how runtime events are observed and how architectural events are combined to form architectural views. In section 2.4.1 an overview was given of the DiscoTect tool [45], which is intended to bridge the gap between runtime events and architectural events. This section focuses on the language used by DiscoTect for the specification of mappings between runtime and architectural events and the advantages and disadvantages of the language.

The DiscoTect tool is based on Colored Petri Nets (CP-nets) [22]. A CP-net consists of a number of *places*, which are attached by directed *arcs* to *transitions*. Each place may contain *tokens*. A token can have data of a certain *type* associated with it. This data allows tokens to be distinguished from each other and is often referred to as the *color* of the token. A transition can remove (*consume*) tokens from *input places* attached to it and put new tokens in attached *output places*. A transition can have an associated *guard expression*, which specifies a condition that must be met by the tokens in the input places of the transition before the transition can *occur*. An occurrence of a transition is *indivisible*: tokens are removed from the input places and placed in the output places at the same time. Multiple transitions (and multiple occurrences of a single transition) can occur concurrently if multiple sets of tokens match the guard expressions of the transitions attached to their places. However, CP-nets are non-deterministic. If multiple transitions can occur concurrently, they are not guaranteed to occur concurrently. One may happen after the other, in any order. This also means that if a token can be consumed by more than one transition at the same time, it is only consumed by one. Which of the competing transitions "wins" is not specified.

To translate runtime events into architectural events, DiscoTect *simulates* a CP-net [45]. Runtime events are represented by tokens, which are placed in input places of the CP-net. The tool evaluates the guard expressions of the transitions and, if possible, executes the transitions, removing the input tokens from the input places and creating new tokens in the output places. Through a number of transitions, the input tokens are transformed into tokens which represent architectural events.

The CP-net simulated by DiscoTect can be specified by the user, using the DiscoSTEP (Discovering Structure Through Event Processing) language. Transitions are specified by rules. Each rule specifies the input and output places of the transition, a *trigger* (guard expression) which specifies when the transition can occur and an *action* which specifies how to create the output tokens, using the data associated with the tokens that matched the

[11] The IDs of the elements cannot be used because Enterprise Architect and the abstraction tools discussed in the following two sections generate different IDs.

```
<call calleeNS="System.ServiceModel" calleeType="ServiceBehaviorAttribute" calleeID="FC4C"
      visibility="public" static="false" constructor="false" calleeOwnerNS="System.ServiceModel"
      calleeOwnerType="ServiceBehaviorAttribute" method="set_IncludeExceptionDetailInFaults"
      returnType="void" timestamp="2119" callerTimestamp="2081" callerID="FBC1">
    <arg name="value" type="bool" value="true" />
</call>
```

Figure 3.6: Example method call event.

trigger. A *composition* connects the output places of one rule to the input places of other rules. An output can be connected to more than one input, in which case each output token is duplicated for each input place. A connection can be *bidirectional*, in which case the tokens are not consumed by the rule which takes them as input, so they can be matched multiple times. Inputs that are not connected to an output are assumed to be provided by the execution tracer. Outputs that are not connected to an input are sent to the architecture builder.

The data associated with tokens is represented in XML format. Triggers and actions are written in the XQuery language. XML schemas are used to declare the types of the tokens. The XML schema of the runtime events written by the TraceProcessor XML output plugin is described in appendix B. An example call event is shown in figure 3.6.

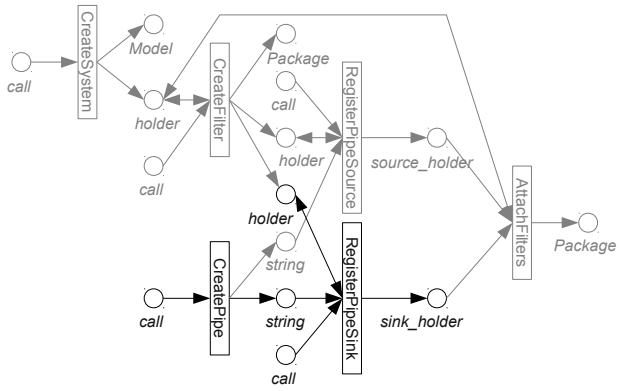
3.3.1 Example DiscoSTEP Rules

Figure 3.7b shows a part of a ruleset that recognizes a very simple implementation of the pipe-and-filter architectural style.[12] In this example, a number of `Filter` objects communicate with each other by calling the `write` and `read` methods of `Pipe` objects. Figure 3.7a shows the CP-net representation of the ruleset. The black parts correspond to the rules shown in figure 3.7b. Each circle represents a place, each rectangle represents a transition. Each place can only store tokens of one type, indicated below the circle in italics.

For each observed method call, a token is placed in all places of type *call*. When a call to the `Main` function of the program is encountered, the `CreateSystem` transition will create a UML model containing a single empty package, in XMI format. Filters and pipes are recognized by the `CreateFilter` and `CreatePipe` rules, which look for calls to the constructors of the `Filter` and `Pipe` classes. To determine which filters are connected to each other, the `RegisterPipeSource` and `RegisterPipeSink` rules monitor which `Filter` instances call the `write` and `read` methods of which `Pipe` instances. Once both methods have been called on a single `Pipe` instance, a connection between the filters that called the methods can be added to the model, for example in the form of a stereotyped UML dependency. This is done by the `AttachFilters` rule.

As discussed in section 3.2.1, the `CreateFilter` and `AttachFilters` rules generate dummy packages which contain UML component instances and UML dependencies, respectively. XMI difference elements are then generated, so that the XMIMerge tool can add the component instances and dependencies to the empty package created by the

[12] It is a simplified version of the PipeFilter example that comes with DiscoTect. The original version can be found at `http://able.fluid.cs.cmu.edu:8080/Able/DiscoTect`



(a) Example CP-net

```
rule CreatePipe {
        input { call $c; }
        output { string $pipe_id; }
        trigger {? $c/@constructor = "true" and $c/@calleeType = "Pipe" ?}
        action {? let $pipe_id := $c/@calleeID ?}
}
rule RegisterPipeSink {
        input { call $c; holder $filter_holder; string $pipe_id; }
        output { sink_holder $sink; }
        trigger {? $c/@method = "read" and $c/@calleeID = $pipe_id and $c/@callerID = $filter_holder/@implId ?}
        action {? let $sink := <sink_holder filterXmiId="{$filter_holder/@xmiId}" pipeId="{$pipe_id}" /> ?}
}
composition PipeFilter {
        CreateFilter.$filter_holder<->RegisterPipeSink.$filter_holder
        CreatePipe.$pipe_id->RegisterPipeSink.$pipe_id
        RegisterPipeSink.$sink->AttachFilters.$sink
}
```

(b) DiscoSTEP rules for the black parts of the CP-net

Figure 3.7: Example CP-net and DiscoSTEP rules.

`CreateSystem` rule and remove the dummy packages. For visual clarity, the places containing these difference elements are not shown in figure 3.7a.

3.3.2 Advantages and Disadvantages of DiscoTect

DiscoSTEP rulesets offer great flexibility. First, they are not fixed to any particular kind of event, allowing support for different implementation platforms and architectural styles. Second, an N:M mapping can be specified between system-level events and architectural events. This allows DiscoTect to handle three common situations:

1. many system-level events may contribute to a single architectural event
2. a single system-level event may indicate multiple architectural events such as the construction of a component and a connector
3. different sets of system-level events can be mapped to the same architectural event, which is important because an architectural concept can often be implemented in several ways

Finally, DiscoTect supports concurrency, because separate architectural events can be recognized from interleaved sets of runtime events. This is possible because a place can contain multiple tokens and a token stays in its place until it is part of a set of tokens which matches a trigger. In the example in figure 3.7a, if multiple filters are reading from and writing to pipes concurrently, `source_holder` and `sink_holder` tokens concerning different pipes and filters can be generated in any order. When a `source_holder` and `sink_holder` with the same `pipe_id` are found, the `AttachFilters` transition will occur, regardless of any other holder tokens that may be present.

While it is easy to specify rules which respond to events that occur at runtime, it is difficult to generate architectural events if certain runtime events do *not* occur. This can cause problems when attempting to identify method calls between architectural elements. In the pipe-and-filter example each instance of an architectural element was represented at runtime by only a single object. This means the `RegisterPipeSink` rule only has to look for a single method call (between objects with known IDs) to identify interaction between filter and pipe instances. In practice however, an instance of an architectural element is typically represented at runtime by several objects which may have different types. It is common for rulesets to recognize only a subset of these objects, particularly in early stages of an architecture recovery effort. In that case, calls between objects that are part of different architectural elements may be indirect, that is, a method that is part of one architectural element may call methods of objects that have not (yet) been recognized as being part of some architectural element, which eventually call methods of other architectural elements. Recognizing such indirect calls between architectural elements is hard, because it is not possible to write a rule which is triggered if and only if a particular object has *not* been recognized as being part of some architectural element.

There are two ways to address this issue, each having some disadvantage. First, recognition can be limited to direct calls. This can lead to false negatives, unless every object is mapped to an architectural element. In practice, completing the mapping manually typically requires a prohibitively large amount of work. Approaches to (semi-)automatically complete the mapping have been proposed in literature [3, 5]. Using such an approach would require combining DiscoTect with another tool.

Second, indirect method calls between objects can be recognized, regardless of whether any of the involved objects has been identified as being a member of some architectural element. This can lead to false positives. For example, if element A calls element B, which in turn calls element C, a call between element A and C will be reported, even if these elements never call each other directly. Such false positives can have a large influence on the results of a modifiability evaluation. Therefore, additional rules have to be written to recognize such false positives and generate events which instruct the architecture builder to remove the false positives. This complicates ruleset development and maintenance.

Besides these disadvantages of the DiscoSTEP language, the current prototype implementation of DiscoTect has poor performance and, as also noted by Ganesan et al. [15], has several bugs. Rather than solving these issues, the decision was made to use Prolog instead of DiscoTect to recognize architectural elements.

```
pipeInstance(Pipe) :-
    className(PipeClass, 'Pipe'),
    instanceof(Pipe, PipeClass).
sink(Filter, Pipe) :-
    filterInstance(Filter),
    pipeInstance(Pipe),
    instanceof(Pipe, PipeClass),
    classMember(ReadMethod, PipeClass),
    methodName(ReadMethod, 'read'),
    methodCall(_, Filter, _, Pipe, ReadMethod, _).
```

Figure 3.8: Prolog version of the DiscoSTEP rules in figure 3.7b.

3.4 Prolog

This section presents a Prolog-based approach for recovering architectural views from observed runtime events, leaving the overall approach outlined in section 2.4.1 largely intact. It is based on existing Prolog-based view recovery approaches [28, 41, 42] and Prolog-based approaches for recognizing design patterns [3, 27]. These existing approaches are discussed in more detail in chapter 6.

A Prolog program consists of a set of *predicates*, each consisting of one or more *clauses*. A clause can be a *fact* or a *rule* which specifies how new facts can be derived from existing facts. For the purpose of architecture recovery, the facts can be generated by the execution tracer. Rules can be written to specify (in terms of facts and other rules) which kinds of runtime behavior and program structure correspond to which kinds of architectural elements. Based on these facts and rules, Prolog can then derive facts which represent architectural elements.

Prolog is a declarative language, allowing the analyst to focus on specifying architectural elements in terms of program behavior and structure, without having to deal with how facts and intermediate results are stored. The advantages of DiscoTect listed in section 3.3.2 can also be achieved with a Prolog-based approach. Different rulesets can be written to support different implementation platforms and architectural styles. An N:M mapping between system-level events and architectural events is possible because a fact can be derived from multiple facts, multiple facts can be derived from the same fact(s) and multiple alternative rules can be written to derive the same fact. Concurrency is also supported. For instance, in the Prolog equivalent of the example pipe-and-filter ruleset (section 3.3.1) shown in figure 3.8, a predicate such as

```
connectedFilters(Source, Sink) :-
    source(Source, Pipe),
    sink(Sink, Pipe).
```

will succeed whenever a filter (source) writes to a pipe and another (sink) reads from it, regardless of the order in which the Filter instances call methods of the Pipe instances.

Contrary to DiscoSTEP rules, Prolog can derive facts from the absence of other facts. This is possible because Prolog makes a *closed world assumption*: if something cannot be proven from the facts and rules known to the Prolog system, it is assumed to be false. This

```
className(class_49, 'System.ServiceModel.ServiceBehaviorAttribute').
classMember(method_86, class_49).
instanceof(obj_FC4C, class_49).
methodName(method_86, 'set_IncludeExceptionDetailInFaults').
parameters(method_86, [['value', 'bool']]).
methodCall(2119, obj_FBC1, method_62, obj_FC4C, method_86, 2081).
parameterValues(2119, ['true']).
```

Figure 3.9: Example extracted Prolog facts.

makes it possible to identify indirect calls between architectural elements, without getting false positives.

A potential disadvantage is that, contrary to DiscoTect, the Prolog-based approach does not support *on-line* analysis, where an architectural view is recovered while the scenario is being executed. This was not found to be a problem in the case study (chapter 4).

Several Prolog systems exist. The XSB logic programming system[13] was chosen because it supports *tabling* [48]. If tabling is enabled for a predicate, XSB will keep a table of answers for each call to the predicate. If a call to a predicate is made multiple times, it only has to be evaluated once. On subsequent calls, the answer is fetched from the table. This can improve performance, but tabling can also simplify ruleset development. For example, consider the following predicate:

```
inherits(Subclass, Superclass) :-
    inherits(Subclass, X), superclass(X, Superclass).
inherits(Subclass, Superclass) :-
    superclass(Subclass, Superclass).
```

This seems correct, but without tabling a query such as `inherits(Subclass, foo)` can result in an infinite loop. With tabling enabled, the query will correctly find all (indirect) subclasses of `foo`. This allows the analyst to focus on how to recognize architectural concepts, rather than on the details of rule execution.

3.4.1 Ruleset Structure

Chapter 4 discusses a case study in which Prolog rules are used to recover architectural views of a system developed by Exact. The rules used in the case study are organized in four layers, where each layer makes use of the facts and rules defined in the layer below it. In addition to these layers, there is a collection of utility predicates which are used by all layers.

Layer 1: Facts This is the lowest layer, consisting of the facts extracted by the execution tracer. Appendix B contains a list of the kinds of facts written by the TraceProcessor Prolog output plugin. As an example, figure 3.9 shows a part of the facts generated for the get-metadata scenario discussed in chapter 4.

[13] `http://xsb.sourceforge.net/`

Layer 2: Recognition Rules The recognition rules are responsible for recognizing architectural elements, relations and their properties, from the facts extracted from the system. Each recognized architectural element has a type and a set of attributes, in the form of key-value pairs. Relations between elements are also represented as elements, with attributes indicating which elements are involved in the relation. For example, an element of type Pipe can have a source and sink attribute referring to the filters that are connected by the pipe.

Two predicates are defined to make the elements and their attributes available to the presentation rules:

- `recognizedElementType(Element, Type)` succeeds if `Element` is an architectural element of type `Type`. In the pipe-and-filter example, all filters can be retrieved with the query `recognizedElementType(Element, filter)`.
- `recognizedElementAttribute(Element, Key, Value)` succeeds if `Element` has an attribute of type `Key` with value `Value`.

For example, using the `connectedFilters/2` predicate shown above, the following rule could be defined to "export" all recognized pipes to the presentation rules:

```
recognizedElementType(pipe(Source, Sink), pipe) :-
    connectedFilters(Source, Sink).
```

Layer 3: Presentation Rules The predicates in this layer determine how the recognized architectural elements should be presented in a view. This is kept separate from the recognition rules to allow the same architectural concepts to be presented in different ways (and different modeling languages), without having to change the recognition rules. In a similar fashion as the recognition rules, the presentation rules define two predicates to represent UML models:

- `umlMetaclass(Element, Metaclass)` succeeds if `Element` is an instance of UML metaclass Metaclass. For example, all UML Components that should be included in the model can be retrieved with `umlMetaclass(Element, 'Component')`.
- `umlElementAttribute(Element, Key, Value)` which succeeds if `Element` has an attribute of type `Key` with value `Value`.

For example, a filter can be represented in UML as an instance of the Filter component:

```
umlMetaclass(filterComponent, 'Component').
umlElementAttribute(filterComponent, name, 'Filter').
umlMetaclass(Filter, 'InstanceSpecification') :-
    recognizedElementType(Filter, filter).
umlElementAttribute(Filter, classifier, filterComponent) :-
    recognizedElementType(Filter, filter).
umlElementAttribute(Filter, name, Name) :-
    recognizedElementType(Filter, filter),
    recognizedElementAttribute(Filter, name, Name).
```

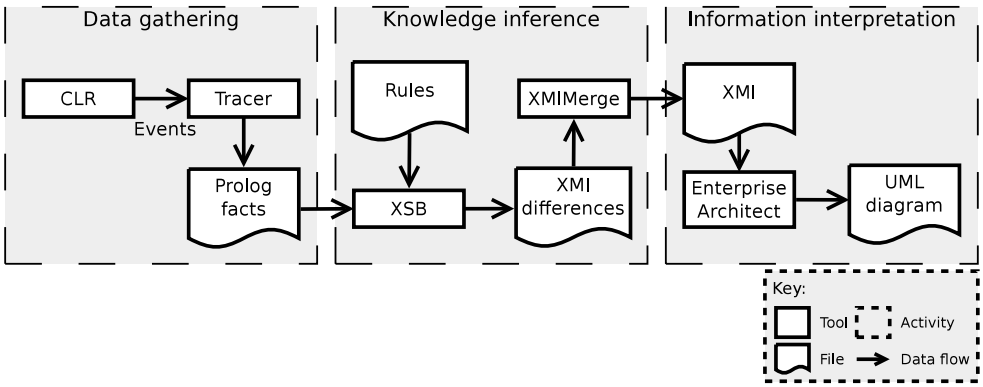


Figure 3.10: Recovery toolset overview.

The first two lines define a UML Component called Filter. The remaining lines define an InstanceSpecification of the Filter component for each filter instance created by the program under analysis.

Layer 4: View Generation Rules The view generation rules export the recovered model for use by the architecture builder. In this case, UML models are written in XMI format. A separate predicate is defined for each supported UML metaclass. Each uses the `umlMetaclass(Element, Metaclass)` predicate to retrieve all instances of a particular metaclass and serializes them in XMI format. Furthermore, the `writeView(Filename)` predicate is defined, which starts the view recovery and outputs the recovered view by calling the metaclass-specific predicates.

Although not strictly necessary, XMI differences are generated and XMIMerge is used to merge them, as with DiscoTect. Alternative approaches are discussed in section 7.1.

3.5 Summary

Figure 3.10 gives an overview of the tools discussed in this chapter, the data flow between them and the activities of the Symphony process in which they are used. Data gathering is supported with an execution tracer based on the .NET Profiling API, which allows tracing the method calls occurring in a running system. The traces can be filtered and exported in a format that can be used by tools which support knowledge inference. One such tool is DiscoTect, but because several problems were encountered with this tool, the choice was made to use Prolog rules (executed by XSB) instead. After processing the output of the Prolog rules with XMIMerge and CopyLayout (not shown due to space constraints), the recognized architectural elements can be visualized in Enterprise Architect, which supports information interpretation.

The next chapter will discuss a case study in which the Symphony process and the tools are applied in practice to recover architectural views for one of Exact's systems.

Chapter 4

Case Study

This chapter discusses a case study in which the recovery tools presented in chapter 3 are used to recover architectural views of a system developed by Exact. The goal of the case study is to determine whether the tools indeed work, providing an answer to research question 1 (section 1.2). To address the second research question, an attempt will be made to validate whether certain architectural decisions have been implemented correctly (for the execution scenario under analysis). The third research question, dealing with the extent to which the recovered views are found readable and useful by the stakeholders of the Exact Connectivity Layer, is addressed in the next chapter.

The following section will introduce the system that will be analyzed in the case study, called the Exact Connectivity Layer. Then, several iterations of the Symphony process will be described, in which the recovery tools were used to recover architectural views of the Connectivity Layer. This is followed by a section on the validation of architectural decisions. Finally, the validity of the case study is discussed.

4.1 Exact Connectivity Layer

The Exact Connectivity Layer is a system that enables other systems to exchange data with several of Exact's systems, such as Exact Synergy Enterprise and Exact Globe. The Connectivity Layer was developed to improve the interoperability of these systems, which was oriented mainly towards batch import and export of data, often using formats that did not follow industry standards. Adding support for different formats or for interactive (near-instant) exchange of data with other systems typically required custom solutions to be developed and maintained, which was relatively expensive because several custom solutions might have to be changed as a result of a single change to a system.

The Connectivity Layer was developed to address these problems. It is used as a layer on top of another system, providing web services which support interactive exchange of data with the underlying system. The web services support Create, Retrieve, Update and Delete (CRUD) operations on *entities*, such as accounts or documents. The Connectivity Layer is based on industry standards such as the Simple Object Access Protocol (SOAP) and hides the details of the underlying system from the outside world.

Most parts of the Connectivity Layer are independent of the underlying system. The system-specific parts are implemented in *providers*. As a result, support for additional systems can be added by implementing additional providers. If the system is changed, only its provider has to be updated, the Connectivity Layer itself and the systems using it can typically remain unchanged. Different providers can implement support for different types of entities. For example, several systems can expose an Account entity through the Connectivity Layer, but providers are not forced to implement support for an Account entity.

The Connectivity Layer was chosen for this case study for two main reasons:

1. It is non-trivial, but at the same time it is not too large and complex to be analyzed in the time available for an MSc project. The Connectivity Layer consists of approximately 15-20 KLOC, mostly written in VB.NET.

2. Development of the system started relatively recently (2008). Most of the original developers are still working at Exact. Furthermore, because the system is relatively new, it is less likely that changes have caused large differences between the as-designed and as-built architecture. This makes it easier to check the correctness of the recovered views.

Furthermore, Exact is interested in improving the interoperability of its systems. Analyzing the Connectivity Layer might support this.

In this case study, the Exact Connectivity Layer is layered on top of Exact Synergy,[1] a web-based business process management (BPM) platform. Among other things, it supports Customer Relationship Management (CRM), document management and workflow management. Synergy consists of approximately 1.5 million lines of code, written primarily in VB.NET and ASP.NET.

4.1.1 Existing Architectural Documentation

The existing architectural documentation of the Connectivity Layer describes the major components of the system and their responsibilities and interfaces. It contains context diagrams showing the Connectivity Layer in the context of external systems and individual components in the context of other components of the Connectivity Layer. Interaction between the components is primarily described with high-level UML collaboration and sequence diagrams. An example of a high-level collaboration diagram, describing the retrieval of entity metadata, is shown in figure 4.1. Furthermore, domain models (describing the data stored and processed by the system) and several design principles and decisions have been documented.

In addition to the architectural documentation, other documentation which could contain information relevant for the recovery of the architecture is available, such as requirements specifications and design documents.

[1] `http://www.exact.com/global/en/products/exact-synergy/index.aspx`

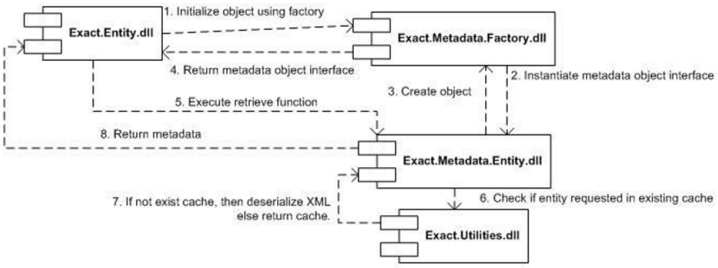


Figure 4.1: Collaboration diagram, from the documentation of the Connectivity Layer.

4.2 Recovery

This section describes the iterations of the Symphony process performed as part of the case study.

4.2.1 Iteration 1

Problem Elicitation

As discussed in section 1.2, if the recovery tools can recover the views found the most useful by the architect when creating the architecture, in such a way that they are readable and useful for a wide range of stakeholders and enable validation of existing architectural documentation, the tools are likely to be able to recover views that are useful in an ATAM evaluation. The goals of the case study are therefore to identify which views are found the most useful and then to actually recover them.

The case study will focus on describing (the interactions between) the server-side components of the Exact Connectivity Layer involved in the following scenario, referred to as the *word-document* scenario:

1. reset the IIS server running Synergy Enterprise and the Connectivity Layer
2. start Microsoft Word, which runs the Synergy Office add-in, which uses the services provided by the Connectivity Layer to, amongst other things, retrieve and store Word documents in Exact Synergy
3. log in to Synergy with the Word add-in
4. download a document from Synergy with the Word add-in

After filtering (which will be discussed later), the trace obtained for this scenario consists of 532717 method calls. This scenario was chosen after some discussions with the architect of the Connectivity Layer, because it is a scenario that could actually occur in practice, but did not appear to be too complex to be analyzed, given the limited time available in a master's project.

Concept Determination

- *Identify potentially useful viewpoints.* Discussions with the architect revealed that high-level collaboration and sequence diagrams are found the most useful. Unfortunately, problems were encountered when importing these kinds of diagrams in XMI format into Enterprise Architect 6.5. In fact, even collaboration and sequence diagrams exported by EA itself were often imported incorrectly. Although it is likely that this has been fixed in newer versions of EA, the decision was made to continue with EA 6.5 and to recover component diagrams first. The components shown in such diagrams are similar to the ones shown in Exact's high-level sequence and collaboration diagrams. Therefore, the recovered components could serve as a starting point for such diagrams in the future. Furthermore, it is relatively easy to generate XMI for component diagrams. To give a rough idea of the interaction between the components, stereotyped dependencies will be used.

- *Define target viewpoint.* Discussion with the architect revealed that many important components of the Connectivity Layer are implemented as *singletons* [14]. A first rough picture of the architecture could be obtained by identifying which singletons are used by which *webservices*. Singletons and webservices will be represented as UML components with respectively a `<<singleton>>` or `<<webservice>>` stereotype. Method calls between components are also shown, represented as UML dependencies with the `<<call>>` stereotype. A component A calls another component B if a method of a class that is mapped to component A directly calls a method of a class that is mapped to component B, or if such a call is made indirectly and all of the methods in between are part of classes that have not been mapped to any component. Multiple calls to the same method are shown only once.

- *Define source viewpoint.* The source view consists of the Prolog facts generated by the tracer (see appendix B).

- *Define mapping rules.* Rules were written that recognize singletons and classes that implement webservices. Additional rules find all (direct or indirect) calls between those components.

- *Determine role and viewpoint of hypothetical views.* No hypothetical views were defined in this iteration.

Reconstruction Execution

Because the word-document scenario is quite large, a simpler scenario will be analyzed first. A demo application (shown in figure 4.2) is used to retrieve the metadata for the Document entity, which involves a single webservice. After filtering out all calls not made to or from (inherited) methods of classes in an Exact namespace, the trace for this scenario (referred to as the *get-metadata* scenario) contains 13000 calls.

Running the ruleset with XSB Prolog, merging the output into a single XMI document with XMIMerge, importing it into Enterprise Architect and manually rearranging the layout

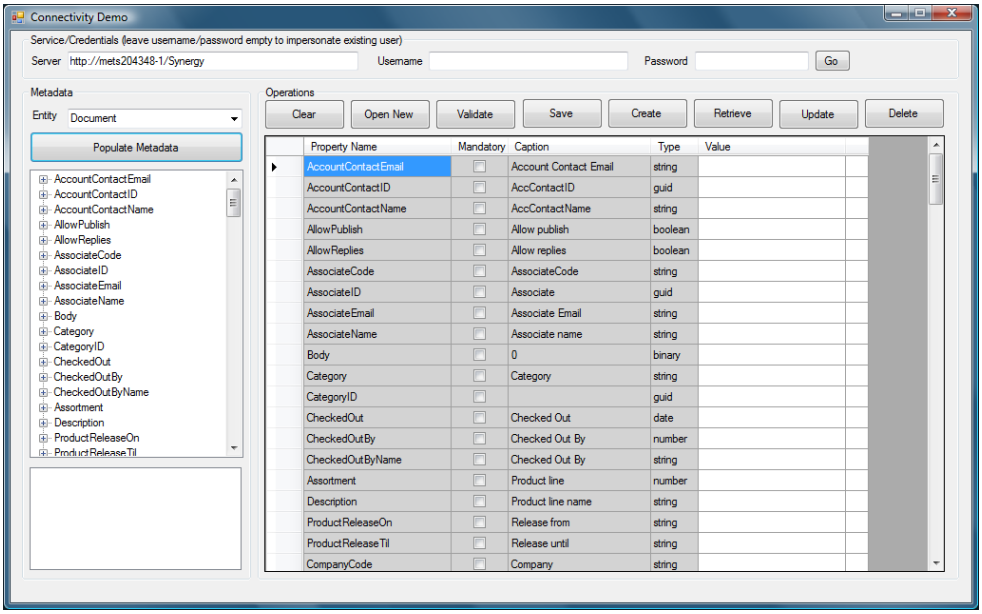


Figure 4.2: Screenshot of the Connectivity Demo application.

results in the component diagram shown in figure 4.3. The diagram shows three components, corresponding to the webservice and the two singletons that have been recognized. In the case of a singleton, the name of the class implementing the singleton is used as the name of the component. The name of the class which acts as a facade for the webservice implementation is taken as the name of the webservice component. In this case, the webservice is called `Metadata`, which makes sense since the scenario involved retrieving metadata for an entity.

The `Cache` singleton calls the `get_CacheProvider` method[2] of the `ServiceLocator` singleton. This method returns an object containing the name of the class which implements the provider and the full path of the assembly in which the class is stored. A Prolog query was used to determine that the call is made from the constructor of the `Cache` class.[3] Looking at the source code of this constructor reveals that, after querying the `ServiceLocator` for the class name and assembly name of the cache provider implementation to use, it passes the names to the `Activator.CreateInstance` method to create an instance of the cache provider. Rules can now be written to recognize this pattern so that other providers can be found automatically.

The `CreateInstance` method (a frontend for the `CreateInstance` method of the `Activator` class in the .NET Framework Class Library) is a generic method which takes a type parameter which specifies a supertype of the class to be instantiated. In the case of

[2]Actually, it is accessing the `CacheProvider` property. Properties are represented internally with `get_` and `set_` methods. These methods will be treated like normal methods.

[3]The tracer uses the C# syntax, where the constructor of a class has the same name as the class itself, rather than the VB.NET syntax, where constructors are called `New`.

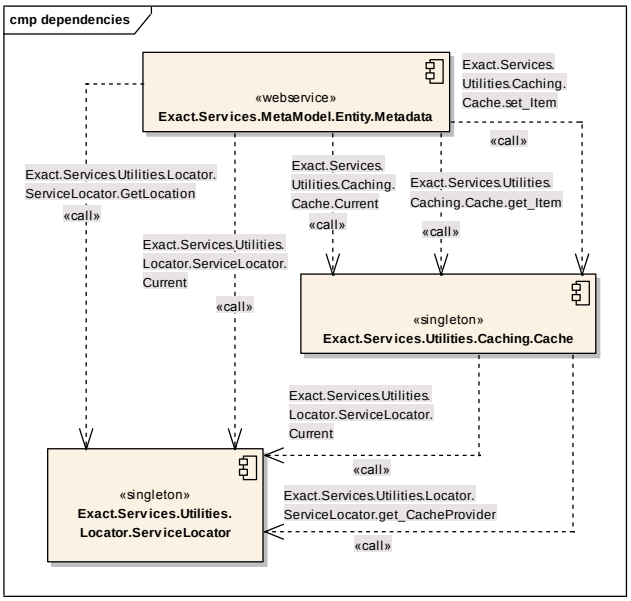


Figure 4.3: UML Component Diagram of the get-metadata scenario in iteration 1.

a provider, this is the interface which must be implemented by the provider. Rules were written which look for method calls to `CreateInstance` and extract the name of the interface and the type of the returned object. This allows provider interfaces and provider implementations to be shown in the view.

Classes of which an instance is returned by the `CreateInstance` method are shown as stereotyped components. Because `CreateInstance` could be called to instantiate other things than providers, the rules check whether the <classname, assemblyname> tuple has actually been returned by the `ServiceLocator`. If this is the case, the `<<provider>>` stereotype is used, otherwise the `<<activated>>` stereotype is used. The interface implemented by a provider is represented as a port and provided interface of the provider component, to show what kind of provider it is. A port and required interface of the same type will be added to each component that calls the provider component. The calls themselves will be represented as dependencies from the required to the provided interface, rather than as direct dependencies between the components (see figure 4.4).[4] Furthermore, a `<<locates>>` dependency is created between the `ServiceLocator` and the provider component. This assumes that the service locator knows the identity of all provider implementations. In the next iteration, this assumption will turn out to be incorrect, leading to a more generic version of the `<<locates>>` dependency.

As discussed earlier, `get_CacheProvider` is used to determine which cache provider implementation to use. The source code of the `ServiceLocator` class was examined to determine how the `ServiceLocator` finds this implementation. The class name and assem-

[4]Currently, this includes all calls, because the information needed to determine if a method is part of a particular interface is not extracted by the tracer.

bly name of the provider implementation are stored in an XML file, which is read using a `StreamReader` and deserialized with the `XmlSerializer.Deserialize` method. This is encoded using Prolog rules which recognize calls to `Deserialize` and determine the name of the deserialized file. This allows the view to explicitly show the location from which the `ServiceLocator` obtains its information and to show any other recognized components which read XML data this way.

Figure 4.3 also shows a call to the `GetLocation` method of the `ServiceLocator`. Looking at the source code of `GetLocation` shows that it uses the same configuration data as `get_CacheProvider`. Prolog queries were used to determine that, given the parameter `"Document"`, `GetLocation` returns the full path of an XML file. This file contains the metadata for the Document entity, for example, it contains the list of properties of a document, as exposed to clients of the Connectivity Layer. Clearly, this is independent of the underlying system, improving interoperability.

The view does not show which methods of the `Metadata` webservice are invoked. To get an idea of the functionality offered by the webservice (and used by the client), these calls should be shown in the view as well. However, calls from the runtime environment to the webservice implementation as a result of requests by the client cannot be recognized directly due to the filtering performed on the trace. Rather than making the filtering criteria less strict (which would increase the tracing overhead and trace size), a "fake" component called `Client` is created. All calls that are (indirectly) made from a method which is itself not in the trace, are represented as calls from this client component.

Finally, to clean up the views, in the next iteration all calls between two components will be grouped into a single dependency. The names of the called methods will be stored in the (Enterprise Architect-specific) Notes field, which can be seen when double-clicking on the dependency in Enterprise Architect.

4.2.2 Iteration 2

For brevity, the reconstruction design phase of Symphony will not be described explicitly for this iteration and subsequent iterations. The changes to the viewpoints and rules have been described in the analysis at the end of the previous iteration. This also reflects the fact that in practice, the information interpretation performed at the end of an iteration sparked the ideas for changes to the Prolog rules in the next iteration.

After changing the ruleset, it was applied to the trace of the get-metadata scenario. The resulting view did not provide any obvious points for further investigation. To learn more about the system, the ruleset was applied to the trace of a more complex scenario, referred to as the *demo-document* scenario: using the demo application, metadata for the Document entity is retrieved, after which information about a document stored in Synergy is retrieved.

The trace obtained with this scenario was filtered more extensively than the get-metadata scenario. Calls from (inherited) methods in classes in an Exact namespace were only included if they were made to (inherited) methods of classes in an Exact namespace or to the methods involved in XML deserialization. Furthermore, methods that are called often but provide little information, such as calls to the constructor of `System.Object` were excluded from the trace. After filtering, the trace contains 59734 method calls.

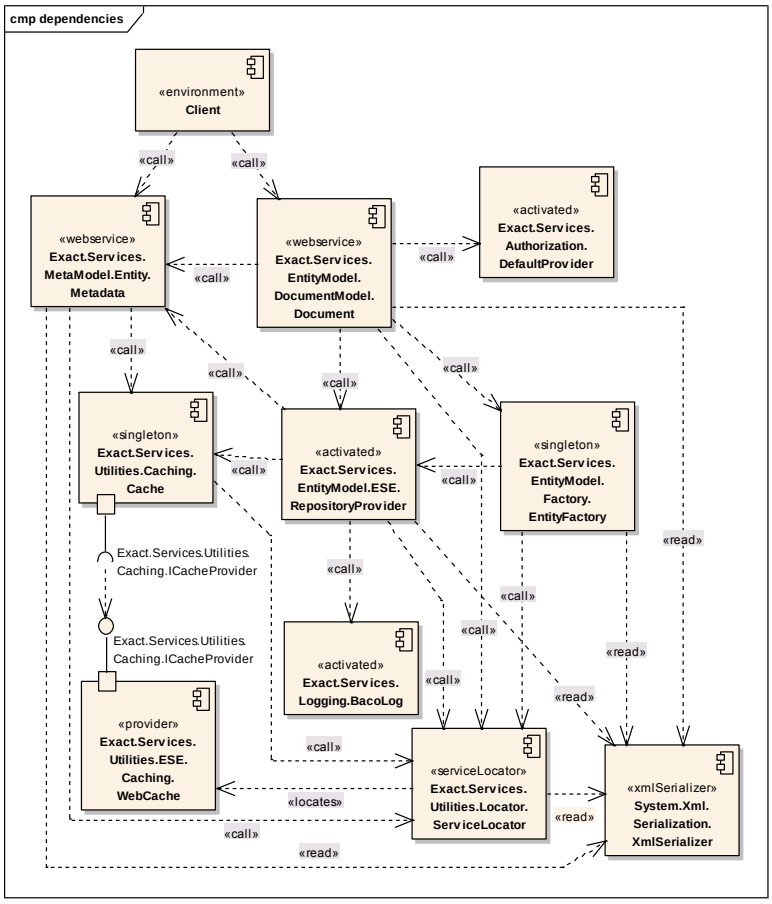


Figure 4.4: UML Component Diagram of the demo-document scenario in iteration 2.

The diagram obtained by applying the ruleset to the trace and manually rearranging the layout is shown in figure 4.4. Several components are recognized, which will be discussed one-by-one below.

`Exact.Services.EntityModel.DocumentModel.Document` is recognized as a webservice component. The architect noted that entity-specific services such as the Document service are automatically generated wrappers around a generic entity service. Clients can either directly use the generic service, or use the entity-specific services, which have a slightly simpler interface. Since the entity engine is an important component, it would be useful to identify it as a separate component. Unfortunately, since the generic entity service is not actually used as a webservice in the scenario under analysis, it cannot be recognized as a webservice with the information that is currently available. It appears to have no other obvious characteristics that could be used to automatically recognize this component. Therefore, in the next iteration, the class implementing the generic entity service will be mapped to a separate component manually. Automatic recognition and the extraction of the necessary information to enable it, are left as future work, discussed in section 7.1.

Examining the source code of the generic entity service reveals that it is essentially a wrapper around the `Exact.Services.EntityModel.EntityEngine` class. All operations on the generic entity are delegated to the `EntityEngine`, which checks whether the client is authorized to perform the action and if so, delegates the action to an `OperationsProvider`. This provider performs the actual action, such as retrieving document data from the underlying system (Exact Synergy Enterprise in this scenario). The `EntityEngine` is hard to recognize automatically based on the available data. In the next iteration it will also be mapped to a separate component manually, leaving automatic recognition as future work.

The list of methods associated with the call dependency from the `Document` webservice to the `EntityFactory` singleton includes the `EntityFactory.GetInstance` method. Prolog queries showed that the call to this method was made from the `GetEngine` method of the generic entity,[5] passing the name of the `OperationsProvider` implementation that should be used by the new entity engine instance created by `GetInstance`. Looking at the source code of `GetEngine` shows that the name is obtained from the service locator, which reads this information from its XML configuration file. This allows the `OperationsProvider` implementation for an entity to be configured at deployment time. As a result, entities do not have a hard dependency to a particular underlying system. Document information can be retrieved from any system as long as an `OperationsProvider` implementation exists for it. Furthermore, new entity types can be added without changing the entity engine itself.

Examining the source code of the `EntityFactory` reveals that it uses the `Activator` to create an instance of the `OperationsProvider` implementation. The only difference with the providers that were recognized earlier, is the way the location of the provider is obtained from the service locator. The service locator only knows the name of the `OperationsProvider` implementation that should be used and the location of the configuration file where the name of the assembly and class containing the implementation can be found. The `EntityFactory` reads this file and then finds and instantiates the provider (in this scenario, `Exact.Services.EntityModel.ESE.RepositoryProvider`). Because of this difference, the provider recognition rules did not recognize the `Logging.BacoLog` class, the `Authorization.DefaultProvider` class and the `ESE.RepositoryProvider` class as provider implementations. However, because the rules did recognize that these classes were instantiated by the activator, they are shown in the view as components with the `<<activated>>` stereotype.

To identify these classes as providers, a more generic way of identifying providers is needed. In both cases the assembly name and class name of a provider are stored in an XML file, which is deserialized, after which the assembly name and class name of the provider are represented as properties of a single object (one object per provider). A few Prolog queries reveal that these properties are set by code in the `XmlSerializer` class. This means that providers can be recognized by looking for calls to `Activator.CreateInstance`, where an instance of a class is returned of which the class name and assembly name have been obtained by reading properties of a single object. The component calling the `XmlSerializer` which set the properties can also be identified, so that a `<<locates>>` dependency can be

[5] Because the generic entity is not shown as a separate component yet, the call is represented in the view as a dependency from the `Document` service.

created between a provider component and the component that determines which provider should be used. This way of deriving the <<locates>> dependency is more generic than the one described at the end of the previous iteration and no longer requires the service locator to be hard coded as the client side of the dependency.

The view shows a dependency between the Authorization.DefaultProvider and the Document webservice. In reality, there is no direct dependency between these components, as several other classes are involved in calls between them. Because dependencies between components can often be identified without a complete mapping of classes to components, there is some flexibility in the mapping of classes to components without reducing the accuracy of the view in terms of the recognized dependencies. When starting out with the analysis of a system, typically only a small portion of the classes is mapped to components. It is important that dependencies are correctly identified in this case, because they can be a valuable starting point for further investigation. Throughout the analysis, the analyst can choose to create a more detailed view by mapping some of the in-between classes to components, or, if the view is considered to be sufficiently detailed, leave the classes unmapped. Both cases occur in the next iteration. The dependency between the Authorization.DefaultProvider and the Document will disappear, because the generic entity and entity engine classes involved in calls between these components are mapped to separate components. There is also an authorization class between the entity engine and the authorization provider, which is not mapped to a separate component, causing the dependency to be shown between the authorization provider and the entity engine. Not creating a separate authorization component results in a more high-level view, while still showing that there exists some dependency between the entity engine and the authorization provider.

4.2.3 Iteration 3

Based on the findings in the previous iteration, the ruleset was modified and a new view was generated for the demo-document scenario (figure 4.5). The providers appear to have been recognized correctly. However, it now appears that the EntityFactory is using the IEntityOperations interface of the ESE.RepositoryProvider. This is correct, because the Activator (which is not recognized as a separate component, but is called by the EntityFactory) is calling the Initialize method of the provider, which is part of the IEntityOperations interface, but in this case, it might be better to represent calls from the Activator with an <<instantiates>> dependency. This will be tried in the next iteration.

The generic entity and EntityEngine are now shown as separate components. The view shows that the EntityEngine communicates with the underlying providers, making the design decision of a single generic entity engine which can handle different types of entities stored in different systems through the use of providers more explicit.

The view looks rather messy. One way to clean up the view is to hide parts of the names of interfaces and components. For example, "ICacheProvider" could be used instead of the full name "Exact.Services.Utilities.Caching.ICacheProvider". To do this, the XMI generator and presentation rules are extended to include an alias[6] for interfaces and

[6] Aliases are Enterprise Architect-specific, they are not part of the UML 2.1.2 standard.

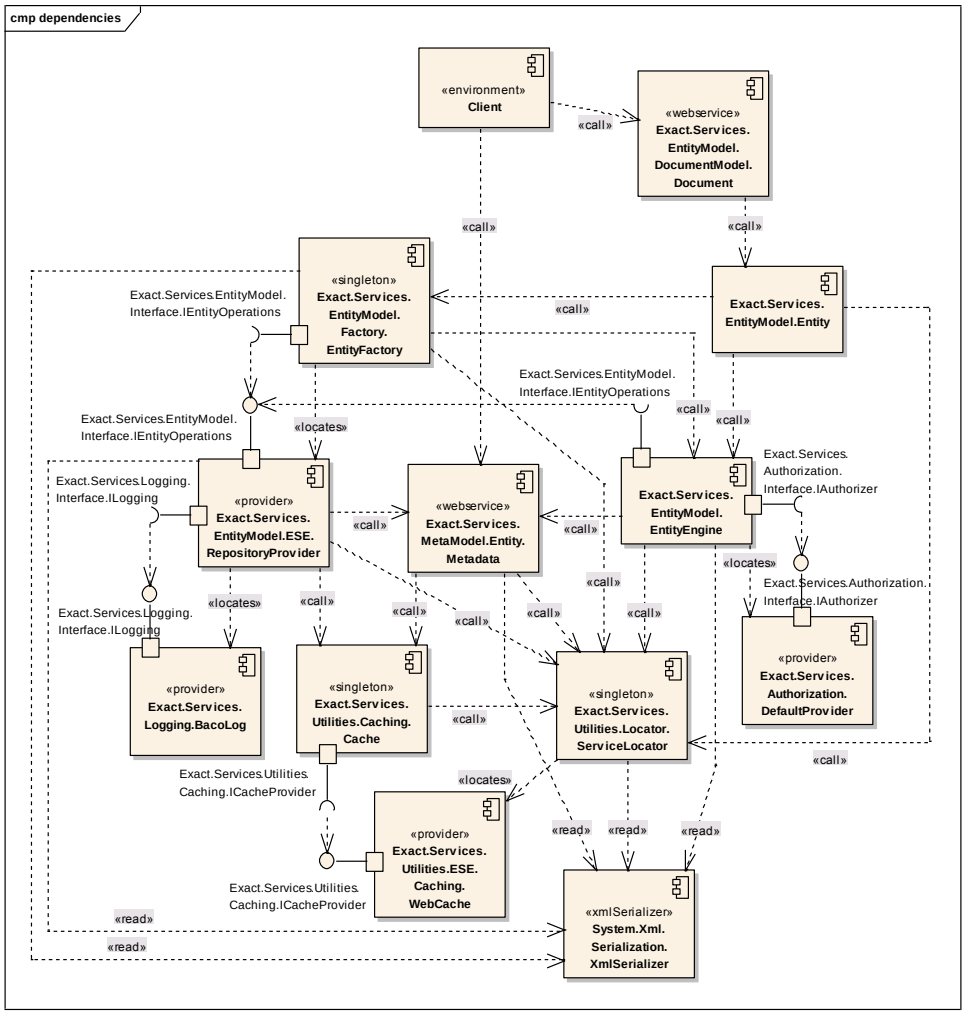


Figure 4.5: UML Component Diagram of the demo-document scenario in iteration 3.

components. The view then only shows the alias, but the full name can still be made visible in Enterprise Architect if needed. For interfaces, the alias only contains the part of the name after the last dot. The alias of components with a name starting with `"Exact.Services."` includes only the part of the name after `"Exact.Services."`.

4.2.4 Iteration 4

The new ruleset was applied to the demo-document scenario. Aside from cleaning up the view, it did not provide any new information. The ruleset was then applied to a trace of the word-document scenario that was the original target for this case study.[7] To obtain this

[7] As was done in the demo-document scenario, to work around a bug in the version of the Connectivity Layer under analysis, the Synergy start page was opened in a web browser after resetting the IIS server.

trace, it was necessary to increase the timeout of the Word Add-in due to the overhead caused by the tracer. The trace was filtered with the same filter as used earlier with the demo-document scenario.

Manually adjusting the layout resulted in the view shown in figure 4.6. The view is quite messy and difficult to read. However, the top of the view clearly shows the client accessing four entity-specific services. As designed, these services all call the generic entity, which in turn uses the entity engine. The entity engine uses different operations providers and authorization providers to handle the requests.

To further clean up the view, each XML file being read will be represented with a separate Artifact. The central `XmlSerializer` component and the many arrows to it will be removed. Each artifact can be placed near the component which reads it, reducing the number of arrows which cross almost the entire diagram.

Currently the diagram provides no indication of the kinds of methods that are being called. To find the names of the methods involved in a call dependency, it is necessary to double-click the dependency in Enterprise Architect, which is often inconvenient. Unfortunately too many methods are called to allow all of them to be shown in the view. Therefore, to provide a rough indication of the kinds of methods being called, at most two methods involved in a call dependency will be shown in the view. The full list of methods can still be seen in the notes. Constructors are never included in the name of a call dependency as these typically provide little information about the functionality of the called component.

The `Profile` webservice shown at the top of the view appears to be different from the other webservices. Clicking on the `<<call>>` dependencies between the client and the webservices shows that, contrary to the other services, the `RetrieveSet` method of `Profile` is invoked. Contrary to the other methods exposed by the entity services, which operate on one entity only (to create a document, retrieve a document, etc...), `RetrieveSet` can be used to retrieve a set of documents which meet certain criteria. From reading the technical design documentation it becomes clear that `RetrieveSet` is implemented separately from the other entity operations. However, its implementation has a structure similar to the way the other entity operations are implemented. There is a separate generic entity (called `Entities`) and a separate engine (called `EntitiesEngine`), which, like the `EntityEngine`, uses providers to do the actual work, although it uses providers which implement a different interface (`IEntitiesOperations` rather than `IEntityOperations`). The generic `Entities` service and the `EntitiesEngine` are manually assigned to separate components, as was done with the generic `Entity` service and `EntityEngine`.

Finally, the view provides almost no information about the system's interaction with its environment. It would be useful to have a *catch-all* component, to which classes are assigned which have not been mapped to any other component. This would include classes in the environment of the Connectivity Layer, such as the underlying system. For example, this allows method calls from providers to the underlying system to be represented as call dependencies to the catch-all component. Calls to the underlying system from a component that is not a provider, which would violate the as-designed architecture, would also be visible, so that they can then be analyzed further. However, assigning all unmapped classes to the catch-all component would also include classes that are part of the Connectivity Layer, but which have not yet been mapped to a component. This could cause a large number of

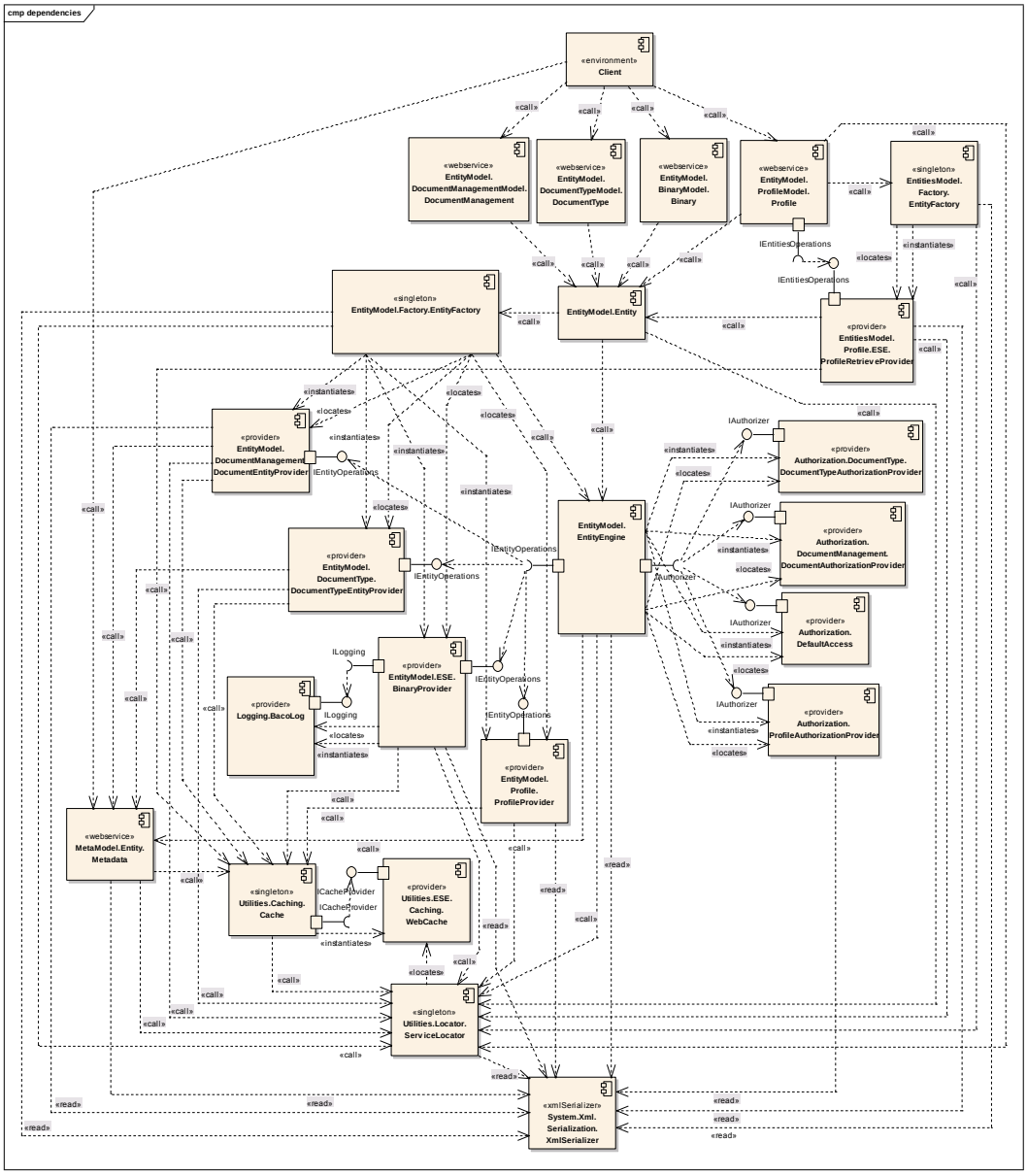


Figure 4.6: UML Component Diagram of the word-document scenario in iteration 4.

dependencies to the catch-all component, making the view hard to read. This could sometimes be useful, because it can point at parts of the system that need further analysis to make the mapping of classes to components more complete. However, this case study will focus on finding dependencies between the Connectivity Layer and its environment, rather than on creating a complete mapping of classes to components. Therefore, a class is only assigned to the catch-all component if none of the classes in its namespace has been mapped to a component.

4.2.5 Iteration 5

While changing the ruleset and testing the changes, it became clear that the catch-all component also included parts of the system that were out of the scope of the case study, such as classes involved in handling the request for the Synergy start page. The ruleset was modified to exclude these parts: recognized elements are only exported to the presentation rules if they are based on (objects and classes involved in) method calls with a sequence number in a given range, specified in terms of markers inserted in the trace after each step in the scenario. Note that the recognition rules do not ignore calls outside this range, so that, for example, singletons instantiated outside the included range, but used within the included range, will still be recognized and exported to the presentation rules. This would have been impossible if the calls outside the range would simply have been removed from the trace.

The new ruleset was run against the demo-document scenario. Many dependencies to the catch-all component were found. However, the catch-all component included many classes that are part of the connectivity layer, but that had not (yet) been mapped to any component. As mentioned earlier, the catch-all component was primarily intended to reveal dependencies to external systems, so classes that are part of the connectivity layer must be removed from the catch-all component. Two things can be done:

1. Identify additional architectural patterns and add Prolog rules to recognize them, or manually assign classes and namespaces to components. For example, many classes which represent entity data also ended up in the catch-all component. Representing them in some way (other than in a catch-all component) could give insight in how data flows through the system.

2. If the level of detail is acceptable, classes and namespaces can simply be excluded from the catch-all component, without mapping the classes to any component. This could save time and effort and result in a cleaner view. For example, many dependencies can be eliminated by excluding classes in the `Exact.Services.Utilities` namespace. If they were represented as a separate component, there would be dependencies between that component and almost all other components. That would make the view very crowded, but provide very little information.

Due to a lack of time, no attempt was made to further analyze the system. Instead, the second option was chosen. The `Exact.Services` namespace was excluded from the catch-all component. Because all classes in the Connectivity Layer are inside (sub-namespaces of) this namespace, the catch-all component now only includes systems in the environment of the Connectivity Layer.

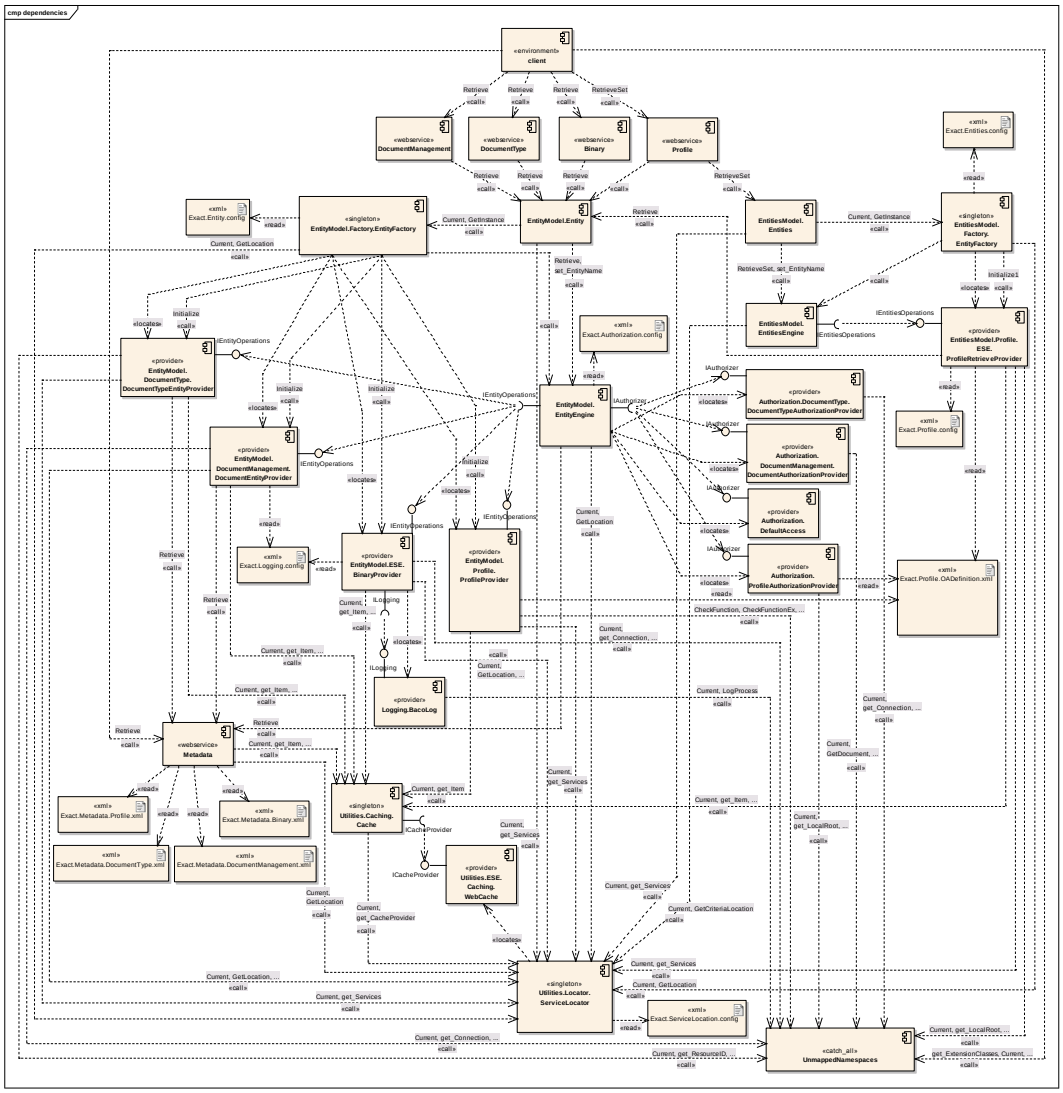


Figure 4.7: UML Component Diagram of the word-document scenario after iteration 6.

4.2.6 Iteration 6

The new ruleset was applied to the demo-document and word-document traces. Since the UML Ports and `<<instantiates>>` dependencies do not really add much information, but do make the view more crowded, they were removed. The resulting view for the demo-document scenario can be seen on page 86. The result for the word-document scenario is shown in figure 4.7.

In this iteration, XSB Prolog was running on a Dell Optiplex 745, which has an Intel Core2 Duo E6300 (1.86 GHz) processor and 2 GB RAM. On this machine, XSB Prolog needs approximately 4 minutes to generate the view for the demo-document scenario and

35 minutes to process the word-document scenario. A processing time of 35 minutes may be acceptable when applying an existing ruleset to a scenario, but it is not acceptable for iterative development of new rulesets. The processing time can be reduced at development time by only using a part of the ruleset. For example, removing the catch-all component reduces the processing time for the word-document scenario to 20 minutes. Another option, chosen in this case study, is to analyze similar but smaller scenarios first. However, this may not always be practical.

The views created in this iteration are used in the next section to determine if the implementation of the Connectivity Layer matches its as-designed architecture and in chapter 5 to assess the extent to which the system's stakeholders consider the views useful.

4.3 Validation of Architectural Approaches

As mentioned in section 1.2, architecture recovery tools can be used to directly recover views for an ATAM evaluation (which will be discussed in the next chapter) and to validate existing views and design decisions. If a recovered view is not found to be readable by the system's stakeholders, it might still allow validation of more readable views. Furthermore, differences between the as-designed and as-built architecture may themselves be relevant in an evaluation.

Unfortunately, as mentioned at the beginning of the case study, existing views such as high-level collaboration and sequence diagrams could not be recovered due to practical problems. This prevents a direct comparison between existing views and the views recovered in this case study. However, it is possible to validate whether certain design decisions have been implemented correctly in the parts of the connectivity layer involved in the word-document scenario. If these decisions influence the system's quality attributes, they could be relevant in an ATAM evaluation, because, as discussed in section 1.1.1, in an ATAM evaluation the effects of architectural approaches and decisions on the quality attributes of the system are analyzed. This allows a preliminary answer to research question 2 (section 1.2) to be given. Providing a more complete answer is left as future work.

Figure 4.7 shows that, except for one dependency, all dependencies to the catch-all component originate from provider components. The only exception is a dependency from the client component, which indicates that it does not pass through any of the other components shown in the diagram. On closer examination this dependency was found to be a false positive. Clearly, all dependencies to external systems have indeed been implemented in providers, as designed. This design decision reduces the *ripple effect*, where components must be modified because another component was changed [2]. It improves the interoperability of the Connectivity Layer, because if an underlying system is changed, only its providers have to be updated. The other components of the Connectivity Layer are not impacted by the change. Similarly, the providers prevent changes to any particular underlying system as a result of changes to the Connectivity Layer, improving the modifiability of the Connectivity Layer.

There are no calls from the metadata webservice to the catch-all component. As designed, the metadata webservice obtains all its information from its own XML files, rather

than from the underlying system. The reason for this is that the interface of the entities exposed by the Connectivity Layer should be the same, regardless of the underlying system. This decision also improves the interoperability of the Connectivity Layer.

The entity-specific services do not directly access any providers. All operations are delegated to the generic entity service and the entity engine. This allows the Connectivity Layer to provide an entity-specific interface, while localizing changes to the entity engine. The former improves the interoperability of the Connectivity Layer. The latter improves its modifiability [2].

The recovered views show that, for the scenario under analysis, the implementation of the connectivity layer closely matches its as-designed architecture. As a result, it is likely that the results of an analysis of the approaches and decisions applied in the as-designed architecture are also valid for the actual system.

In order to find directions for further improvement, it is also interesting to look at design decisions that could not be seen in the view. Several decisions might be made visible by recovering a more dynamic view, such as a high-level sequence or collaboration diagram as originally intended. For example, it could show whether authorization checks are performed on every entity access occurring in the scenario, which is relevant for the security of the system.

The decision to implement fairly basic CRUD (Create, Retrieve, Update, Delete) operations rather than higher-level services influences interoperability in several ways. Making this decision visible in the view is not possible with dynamic analysis alone. Although a scenario involving all CRUD operations could be visualized, this does not exclude the possibility that other operations have been implemented as well. Combining static analysis and dynamic analysis could provide a solution for this problem.

4.4 Validity Threats

The validity of the case study presented in this chapter is affected by threats to construct, internal and external validity. *Construct validity* refers to whether the measures used match the concepts being studied, *internal validity* deals with whether inferences are correct and *external validity* refers to the extent to which findings can be generalized beyond the studied case [52].

4.4.1 Construct Validity

The goal of the case study was to recover the views found must useful by the architect when creating the architecture. Due to practical problems with importing these views into Enterprise Architect 6.5, these views could not be recovered. Instead, a different kind of view was recovered, which means the case study assessed the suitability of the recovery tools for a different purpose than required, which limits the construct validity of the case study. Therefore, the case study should be considered as no more than a preliminary evaluation of the tools. A future case study will have to determine whether the recovery tools, with a newer version of Enterprise Architect (or a different visualization tool), can recover the required views.

4.4.2 Internal Validity

The case study focused primarily on the active components involved in the scenario and did not consider the data passed around. This could threaten the internal validity of the case study, because it might hide dependencies from data classes to the underlying system as dependencies from the client to the catch-all component, making them harder to spot. Failing to recognize a dependency might incorrectly lead to the conclusion that a system correctly implements its as-designed architecture. This was not the case in the case study, but this had to be verified manually, which is not a scalable solution. Furthermore, the view provides no clear clues that these classes were not included. A future case study could focus on finding an easy way to include them in the view.

4.4.3 External Validity

External validity is threatened by several factors. First, the case study made use of similar, but smaller scenarios than the target scenario to simplify the initial stages of the analysis. There is no guarantee that this is always possible in practice.

Second, the system under analysis was relatively new and the architect of the system was available to answer questions about the architecture. His suggestion to look for singletons to identify important components was very helpful. Unfortunately, such information may not always be available.

Third, it is unclear to which extent the complexity of the ruleset and the size of the trace are representative of cases encountered in practice. In practice, scenarios are likely to be larger than the word-document scenario. Furthermore, as is shown by the reduction in processing time when removing the catch-all component, there can be large differences in the complexity of the rules. More work is needed to determine the scalability of the approach in terms of the complexity and size of these inputs.

Finally, no components of the underlying system (Exact Synergy) were identified automatically. Clearly, this system is using different kinds of components and/or a different way of implementing them than the Connectivity Layer. This made it easy to focus on the Connectivity Layer, but it is not safe to assume that this will be the case for all systems.

Chapter 5

User Study

The involvement of the system's stakeholders in an ATAM evaluation is crucial [24]. Although it may seem that this is mostly limited to the scenario elicitation steps, in practice the stakeholders are involved throughout the analysis to make sure their concerns are sufficiently addressed [2]. To allow active participation of the stakeholders, the views presented during the evaluation should (at least) be readable and usable for the stakeholders involved.

One way to evaluate whether the recovered views are considered useful in an ATAM evaluation would be to actually use them in an ATAM evaluation. However, performing an ATAM would require a relatively large amount of effort and time. Furthermore, a thorough evaluation would require the recovery of different kinds of views and possibly performing more than one ATAM evaluation. Even if this would have been feasible, one might question whether such a large amount of effort is justified without having performed some preliminary tests first. Such a test, providing a preliminary answer to research question 3 (section 1.2) is discussed in this chapter. Although it will not prove whether views recovered with the recovery approach discussed in this thesis are useful, it is intended to find directions for future work in this area. The evaluation focuses only on the recovered views, the recovery process itself is not evaluated. For example, assessing the difficulty of creating, maintaining and applying the ruleset is left as future work.

5.1 Evaluating Usefulness

To assess whether the views recovered in the case study in chapter 4 can be read, understood and used by different groups of stakeholders, a survey was conducted. In this survey, a questionnaire was sent (by e-mail) to representatives of different groups of stakeholders of the Exact Connectivity Layer. The questionnaire presented a recovered view, followed by several questions to assess whether the participants consider the view useful in practice. The filled out questionnaires were returned by e-mail.

Another possibility could have been to interview the stakeholders. An interview would allow a more thorough discussion of the view, during which opportunities for improvement could be identified that might otherwise have remained unnoticed. However, interviews also have some practical problems. For instance, interviewing representatives of all groups

of stakeholders face-to-face is difficult, because some are located in The Netherlands and others in Malaysia. Furthermore, there is a greater risk that interviewer effects will influence the results [4]. For example, people might be more reluctant to give negative feedback if an interviewer (who is also the person who created the view that is being evaluated) is present. It is also harder to ensure that the same questions are always asked in the same way and the same order and that all participants receive the same information. Answering questions that might be asked by the stakeholders during the interviews could give some stakeholders more information than others, which could inadvertently influence the results. Finally, an interview would typically use more open questions, making it harder to compare the answers given by the different stakeholder groups.

5.2 Survey Design

This section will discuss the rationale behind the survey questions. To allow an easier comparison of the answers given by the different stakeholder groups, all participants were asked to complete the same questionnaire. The questionnaire is included in appendix C.

The questionnaire is sent to the participants, who are asked to fill out the questionnaire by themselves. Where applicable, the recommendations given in [4] for such questionnaires have been applied. To reduce the risk of a low response rate, the questionnaire was kept short and where possible, closed questions were used rather than open questions. Another advantage of closed questions is that the answers are easier to process and compare. The participants are given the opportunity to provide additional comments and remarks, so that answers can be given that are not covered by the multiple-choice options. This feedback could support the identification of areas on which to focus future work, which is one of the goals of the survey.

The questionnaire starts out with an introduction, which briefly motivates architecture evaluation and recovery and explains the goal of the survey. The introduction is followed by several questions about the background of the participant. The participants are asked to rate their own knowledge of (or experience with) UML Component Diagrams and the Connectivity Layer, on a scale from 1 (no knowledge/experience) to 5 (in-depth knowledge/experience). These questions are included because the background of the participants could differ from participant to participant, even within the same stakeholder group. It could be interesting to determine if there is a correlation between the background of the participants and the extent to which they consider the view to be useful. For example, if the view is only found readable by a group of participants with a specific background, its use in an ATAM may be limited. Furthermore, the answers to these questions allow assessing whether the background of the survey participants is sufficiently diverse. For example, if all participants consider themselves to be experts on the architecture of the Connectivity Layer, the survey results provide little indication about the use of the view in an ATAM evaluation in practice, which typically also involves people who are not experts on the architecture.

Following the questions about the background of the participants, the recovered architectural view is presented, accompanied with a short discussion of the view. Finally, the actual evaluation questions are listed.

5.2.1 Evaluation Questions

The list of evaluation questions consists of 7 main questions, which are discussed briefly below. The first question consists of 5 statements, which the participants are asked to rate on a Likert-scale [4] ranging from 1 (strongly disagree) to 5 (strongly agree). The statements deal with the extent to which the components and relations shown in the view are recognizable and correspond to the concepts the participants normally use to reason about the Connectivity Layer. If the components and relations correspond to concepts the stakeholders are familiar with, it may be easier for the stakeholders to see how (the analysis of) the view is related to their concerns, enabling more active participation.

One of the statements aims to assess whether the textual description accompanying the view made it easier to understand the view. A consistent high rating for this statement could indicate that the view alone does not provide enough information. This means considerable effort may have to be spent to interpret and explain a view after it has been recovered, limiting for example the ease with which an existing ruleset can be reused to recover views for different (but similar) systems. Ideally, such a statement would not be rated directly, but alternative approaches were not feasible. For example, if an experiment involving multiple groups of participants would have been carried out, it would have been very hard (if not impossible) to make sure that differences in the backgrounds of the members of the different groups would not influence the results.

Second, the participants are asked to determine if the view is correct. Several participants have in-depth knowledge of the architecture, design and/or implementation of (parts of) the Connectivity Layer. This allows at least a cursory check of the correctness of the view by several persons.

In the third part, the participants are asked whether they consider the view to be useful. In order to be useful, the view must be understandable and must allow the participants to reason about the system. Therefore, if the participants find the view hard to understand, they can be expected to consistently give a low rating. The results must be interpreted carefully though. Low ratings do not necessarily mean that the views are hard to understand and high ratings do not necessarily mean that the views cannot be improved further. To assess whether a view is useful for a particular activity, it might be better to measure differences between two groups of participants, where only one group has access to the view while performing the activity. This approach was not chosen because it would require more time and effort than is available for this (preliminary) evaluation and, as mentioned earlier, it would be very difficult to form multiple groups that are similar in terms of the background of the stakeholders.

Again using a Likert scale from 1 (not useful at all) to 5 (very useful), the participants were asked to rate the extent to which they consider the view to be useful for each of four activities: adding a new feature, bug fixing, effort estimation and finding ways to improve the quality of the system. Because all participants receive the same list of activities, a separate “no opinion” option is provided in case the participant cannot rate the use of the view for a particular activity (for example, because (s)he never carries out that activity). Not all of the listed activities are directly related to architecture evaluation. However, the activities are chosen such that, to be useful for any of the activities, the view must provide

some insight into the structure of the system and the way the system works. Furthermore, the list of activities allows the participants to consider the use of the view for a particular purpose. This is needed, because like most tools, views are often useful for some purposes and less useful for other purposes, making it hard to rate the usefulness without referring to a particular purpose. The participants were also given the opportunity to suggest additional applications for the view.

While considering the use of the view for a particular purpose, the participants may find that certain parts of the view are irrelevant, or that important information is missing. Questions 4 and 5 ask whether this is the case and, if so, what should be added or removed.

Question 6 asks whether the view would be useful for a novice team member. This would require the view to present key abstractions of the system in a way that is understandable for people with little or no prior knowledge about the system. If this is the case, the view could be useful when experts who have not been involved in the development of the system are asked to participate in an evaluation.

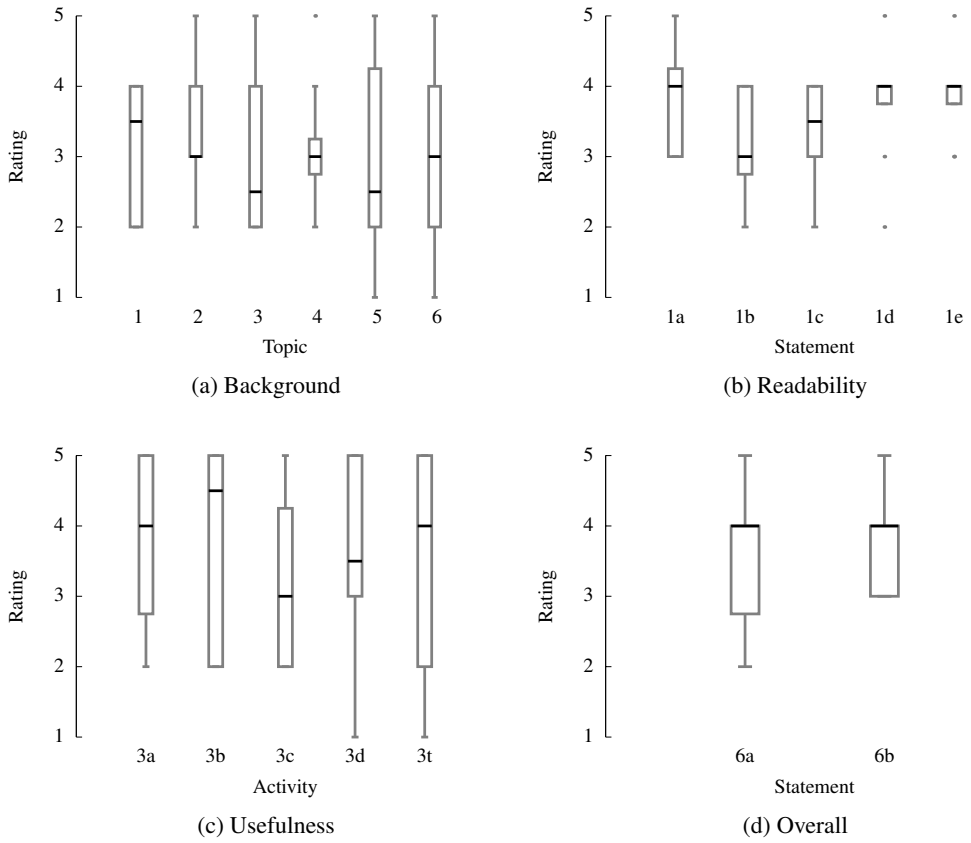
Finally, the participants can give an overall rating of the view and are given the opportunity to give any additional remarks and feedback they like.

5.3 Survey Participants

A group of 13 people, all employees of Exact, were asked to participate in the survey. The group was selected by the lead architect of the Connectivity Layer and includes representatives of the following groups of stakeholders:

- *Software architects team.* Includes the architects who have been involved in the design of the architecture of the Connectivity Layer and Exact Synergy.
- *Development.* Includes people who have been involved in the implementation of the Connectivity Layer and Exact Synergy.
- *Product management.* Among other things, product management is responsible for eliciting and defining the requirements of Exact's systems.
- *Research team.* Members of the research team are experts on particular topics, but have not been directly involved in the development of the Connectivity Layer.
- *Exact Online.* Exact Online is a system developed by Exact that currently does not use the Connectivity Layer to communicate with other systems. If the Connectivity Layer is added to Exact Online in the future, it may have to meet different quality attribute requirements than in the context of Exact Synergy. An ATAM evaluation could be a useful tool to determine if the Connectivity Layer can potentially meet those requirements and what risks might exist [2]. Such an evaluation would certainly involve representatives from the Exact Online team.

Representatives of these groups of stakeholders would typically participate in an ATAM evaluation in practice.



(a) Background (b) Readability

(c) Usefulness (d) Overall

Figure 5.1: Box-plots of the response to the Likert-scale items.

5.4 Analysis of Results

This section discusses the results of the evaluation of the view recovered for the demo-document scenario after iteration 6, as described in section 4.2. Unfortunately, there was insufficient time to evaluate the view recovered for the word-document scenario.

The participants were asked to complete the questionnaire shown in appendix C, which includes the recovered view. The Enterprise Architect file containing the recovered view was not given to the participants, so differences in the level of experience with Enterprise Architect could not influence the results. However, it also limited the evaluation to the directly visible parts of the view. For instance, the participants could not retrieve additional information about dependencies, which they could do in a real ATAM evaluation.

Out of the 13 people who were asked to participate, 8 people responded. This included one member each of the Research and Exact Online teams and two members each of the Software Architects, Development and Product Management teams. Appendix D lists the answers and comments given by the participants. These results will be discussed in the remainder of this section.

Figure 5.1a shows a box-plot of the answers given to the questions concerning the background of the participants. Clearly, there is no topic on which all participants consider themselves to be experts and no topic on which all participants consider themselves to be non-experts. On all topics, at least two participants rated their expertise higher than 3 and two participants rated their expertise lower than 3 (except for the functional requirements topic, for which only one participant rated lower than 3). This means the group is sufficiently diverse for this preliminary evaluation.

5.4.1 Readability

A box-plot of the answers to the statements regarding the readability of the view is shown in figure 5.1b. None of the participants rated statement 1a lower than 3. This means that none of the participants found the components and relations shown in the view completely unrecognizable. However, given that almost half of the participants answered 3 suggests there is room for improvement.

Out of 8 participants, 6 answered 3 or higher to statement 1b, suggesting that at least for some stakeholders it is indeed possible to recover architectural concepts with which they are familiar from an execution trace. However, the maximum rating of 4 and a median of 3 also show that further improvement is necessary.

Two participants, both product managers, answered 2, meaning that they do not agree that the elements and relations shown in the view correspond to the concepts they normally use to reason about the system. It is interesting to determine why this is the case. One possible reason is that their concerns are different from those of the other stakeholders, requiring different views, possibly representing the system in terms of different abstractions. Determining whether this is the case and if so, whether the required views can be recovered or validated with the recovery approach and tools presented in this thesis, would be an interesting topic for future work.

The participants tend to agree that the view contains too many elements and relations (statement 1c). Considering the fact that this view (page 86) is far less crowded than the view based on the word-document scenario that was the original target for the evaluation (figure 4.7), this suggests that the view might not scale well enough. To confirm this, the word-document view would have to be evaluated. Although the word-document scenario was chosen because it could actually occur in practice, it is by no means the most complex practical scenario imaginable. This suggests that an interesting topic for future work is the use of the recovery approach and tools to recover higher-level abstractions of the system.

Most participants agree that the description accompanying the view made it easier to understand the view (statement 1d). This indicates that some amount of expert interpretation of the recovered view remains necessary. One respondent commented that describing the purpose of each component is also important for some activities. This is currently not supported by the recovery tools and would have to be done manually by an expert. The result is that the amount of effort needed to apply an existing ruleset to similar systems (or possibly, to newer versions of the same system) is increased, further reducing the reusability of rulesets. This makes the improvement of the reusability of the rulesets another interesting topic for future work.

5.4.2 Correctness

Half of the participants replied that they could not determine whether the view was correct. These respondents were mostly people who had limited involvement in (the technical aspects of) the Connectivity Layer. One respondent, the lead architect of the Connectivity Layer, noted that the view was incorrect because it was missing one component. This component was not covered by the ruleset, because it is not a singleton, webservice, provider or manually mapped component. In practice such a finding would most likely lead to additional architecture recovery iterations. However, it also indicates a risk. Currently, reviews by experts are the only way to determine whether the view is correct. For various reasons, experts may not always be available in practice, in which case problems are likely to remain unnoticed.

5.4.3 Usefulness

Figure 5.1c summarizes the extent to which the participants found the view useful for specific activities: adding a new feature (3a), bug fixing (3b), effort estimation (3c) and finding ways to improve the quality of the system (3d). Box-plot 3t summarizes all ratings of all activities combined. The medians of the ratings are at least 3, indicating that most of the participants found the view at least slightly useful for the activities considered. Several additional uses of the view were suggested: “dynamically generated documentation”, “as architecture for a similar system” and “security analysis”. All of these suggestions are related to architecture evaluation, but could also support other activities.

High ratings and suggestions for additional applications of the view were given by members of different stakeholder groups. This suggests that the view is indeed accessible to a diverse group of stakeholders, as intended. However, the numbers also indicate that there is room for improvement. In particular, the activities that are most closely related to architecture evaluation (3c, 3d and to some extent 3a) received the lowest ratings. Overall, the answers indicate that architecture recovery could also be useful to support other activities within Exact than architecture evaluations considered so far.

With a median of 3.5, most participants do not disagree that the view could be useful to find ways to improve the quality of the system (3d). One participant did not find the view useful for this purpose at all (rating 1). Interestingly, the other participant in the same stakeholder group found the view very useful for this activity (rating 5). Clearly, this needs further investigation.

The architects gave slightly higher ratings than the developers, particularly for the feature addition and bug fixing activities, which suggests that the view addresses the concerns of the architects better than the concerns of the developers. This matches a comment made by one of the developers in response to question 7, stating that fixing bugs requires more details. This could indicate that the recovery tools can indeed be used to recover abstractions at the architectural level, as perceived by the system’s stakeholders, but additional research is needed to determine more conclusively whether this is the case. If so, it may also be interesting to determine whether different views are needed to address the concerns of the developers in an ATAM evaluation, although the differences between the answers given by

architects and developers were smaller for the activities most closely related to architecture evaluation (3c and 3d).

5.4.4 Completeness

75% of the participants stated that the view contains all elements and relations they consider essential (question 4). In response to question 5, 62.5% of the participants stated that the view does not contain elements or relations they consider irrelevant. Overall, the level of abstraction of the view appears to come close to what the stakeholders need. However, as discussed earlier in the analysis of the answers to the readability-related questions, the view may not scale well enough. Some stakeholders suggest removing certain elements (such as the XML files or the service locator), which would certainly clean up the view. It remains to be seen whether removing these elements allows the view to represent scenarios larger than the word-document scenario in a useful way. There is also a possibility that removing these elements will remove information that is essential for other stakeholders.

Other participants suggested adding things to the view, in response to question 4 and question 7. In particular, database access is mentioned as something that should be shown more explicitly in the view. Currently, this is effectively collapsed into the catch-all component, making it invisible. Subsequent iterations of the recovery process could be applied to further refine this part of the view. However, this would also make the view larger and more complex, which could make the scalability problem worse for large scenarios.

Future work could attempt to perform additional iterations to refine the view. Another option is to search for different abstractions, which scale better while still addressing the needs of the stakeholders. One respondent mentioned that the different tiers or layers in the architecture were not visible. In the current approach these could be made visible by assigning component types to tiers. This would also allow hiding components in tiers that are not relevant for a particular purpose. In response to question 7, respondents also suggested creating different views at different levels of abstraction and allowing the user to zoom in from a high-level view to a more detailed view. Such functionality could significantly improve scalability. Adding it to the recovery tools will briefly be discussed in section 7.1.

5.4.5 Overall Ratings and Feedback

Figure 5.1d shows a box-plot of the extent to which the participants consider the view useful for novice team members (6a) and whether they agree that overall, the view is useful (6b).

Most stakeholders agree that the view is useful for novice team members. As discussed in section 5.2.1, this indicates that the view could be used to describe and explain the architecture to people with limited prior knowledge of the system, such as external experts participating in an ATAM evaluation.

Most participants agreed (and none disagreed) that overall, the view is useful. Most participants rated the overall usefulness with 3 or 4, which is encouraging, but also indicates that further improvements are desirable. Many issues that still need to be addressed were identified throughout the analysis. The overall mildly positive response of the participants

to the current view indicates that future work on these issues could be worthwhile. The recovery process and tools applied in this thesis do not appear to be a dead end.

Several respondents included additional comments. Some of them have been discussed earlier, the remaining ones are discussed here. One participant commented on the fact that a fairly new system was analyzed. Analyzing an older system of which less is known could indeed provide more interesting information, about the system itself, but also about whether the recovery process and tools can indeed be used to recover architectural views of an old system in practice. However, the fact that less is known about older systems also makes it harder to check whether the recovered views are correct. For this reason, a newer system was chosen, leaving analysis of an older system as future work.

Finally, one participant suggested the use of additional tools to support impact analysis of changes to the system. This could be useful in several ways, for example to compare alternative solutions, or, in an ATAM evaluation, to support the analysis of scenarios which represent modifiability requirements.

5.5 Validity Threats

This section will discuss issues which might affect the validity of the conclusions drawn from the survey results.

5.5.1 Construct Validity

Because the survey was intended to give only a preliminary indication of the usefulness of the recovered view in an ATAM evaluation and to find directions for future work, no hard conclusions were drawn about the usefulness of the view. However, in order to be able to draw *any* conclusions from the results, it is important to determine to what extent the survey is representative of the situation in a real ATAM evaluation.

First, only one view was evaluated, whereas multiple views are typically used in an ATAM evaluation. Furthermore, this view was not in the initial list of views found to be most useful by the architect. Although the case study and survey results show that it is possible to recover a view that, to some extent, is found readable and useful by the system's stakeholders, more work is needed to determine if the views found most useful in ATAM evaluations in practice can also be recovered.

Second, the activities for which the usefulness of the view was rated do not completely represent the activities in an ATAM evaluation. Therefore, the usefulness ratings obtained should not be considered to indicate usefulness in an ATAM, but rather to indicate that the view is not found to be totally incomprehensible. The effort needed to develop rulesets was not evaluated at all, although it is essential in practice.

Finally, the survey did not attempt to simulate the circumstances of an ATAM evaluation. In an ATAM evaluation, the view is often discussed by a group of stakeholders, where a stakeholder could easily influence the opinion of other stakeholders. Furthermore, the participants in the survey did not have access to all information in the recovered view.

5.5.2 Internal Validity

Two main factors influence the internal validity of the survey. First, to determine if the view can be used by a wide range of stakeholders, the background of the participants must be sufficiently diverse. Based on the answers given to the questions concerning the background of the participants, it appears that there is no topic on which all participants are experts and no topic on which all participants are non-experts. Although there are topics for which the number of experts is not exactly equal to the number of non-experts, the group is considered to be sufficiently diverse, at least for a preliminary evaluation.

However, the way the background of the participants is determined is not without problems. For instance, the possible options (such as "in-depth knowledge") are not sharply defined and different participants may have different ideas about when to choose which option. This threatens the validity of conclusions based on these ratings, but more objective ways to characterize the backgrounds of the participants were not feasible in this study.

Second, the participants might feel pressure to give desired answers, such as giving higher ratings. Reducing this threat was one of the reasons why face-to-face interviews were not used. Furthermore, the introduction to the questionnaire explicitly states that the personal opinion of the participants is what is important and that as such, there are no right and no wrong answers. However, this kind of influence can not be completely excluded.

5.5.3 External Validity

The external validity of the survey is influenced by several factors. First, although the system under analysis was not trivial, systems analyzed in practice may be larger and more complex. As one respondent commented, the system was new and much was already known about it. More work is needed to determine if the results generalize to larger systems and systems about which little is known. Analyzing larger traces is particularly important because it is suspected that the recovered view is not sufficiently scalable.

Second, only one view was evaluated. It seems likely that other views with similar abstractions can also be recovered. However, based only on the results for this view, it is impossible to determine whether the approach outlined in this thesis can be used to recover completely different kinds of views, which may be needed for an ATAM evaluation.

Third, several method names, such as `Retrieve`, give a good indication of the purpose of the method. In some cases, this may have helped to make the meaning of `<<call>>` dependencies clear. Unfortunately, it is certainly not guaranteed that all systems will have descriptive method names. Furthermore, only a few method names can be shown in the view. Whether the list of names sufficiently describes a call relation between two components depends on the scenario and the system under analysis.

Finally, in practice an ATAM could involve other kinds of stakeholders than those participating in the survey. Although the diverse background of the respondents might suggest that the results generalize beyond the groups of stakeholders participating in the survey, it is still possible that the view does not contain the information necessary to address the concerns of other stakeholders. Furthermore, each group of stakeholders is represented by only 1 or 2 participants, who sometimes give very different ratings.

Chapter 6

Related Work

A large amount of work has been done in the areas of architecture evaluation [1, 11, 44] and architecture recovery [39]. This chapter focuses on (1) work combining architecture recovery and evaluation, (2) architecture recovery using dynamic analysis and/or pattern matching techniques and (3) the evaluation of the usefulness of recovered views in practice.

6.1 Combining Architecture Recovery and Evaluation

This thesis discusses an architecture recovery approach to support an ATAM evaluation, applying architecture recovery and evaluation methods iteratively, similar to the approach taken in [43]. Störmer proposes the SQUA^{3}RE (Software Quality Attribute Analysis by Architecture Reconstruction) approach to combine architecture evaluation and recovery [47]. A SQUA^{3}RE evaluation involves making models of the architecture, which are used to estimate the effects of changes to the architecture on the system's quality attributes. The information needed to construct and evaluate the models is obtained by formulating queries, which are answered using architecture recovery techniques. The process is highly repeatable and can often be automated to a large extent. The use of models makes SQUA^{3}RE primarily suitable for quantitative analysis, whereas ATAM also allows a more coarse-grained qualitative evaluation. The architecture recovery techniques and tools discussed in this thesis can also be used in a SQUA^{3}RE evaluation. Furthermore, the methods can be combined, for example by starting out with (scaled-down) ATAM evaluations and, as the architecture and quality attribute requirements are better understood, switching to the SQUA^{3}RE approach, allowing more automation and a more detailed and efficient evaluation.

Gorton and Zhu [16] evaluate several tools for the purpose of recovering architectural views just-in-time for an architecture evaluation. They focus on evaluating the modifiability of a system and emphasize the use of metrics to narrow down the scope of the analysis to parts of the system that are potentially hard to modify. A disadvantage of using metrics alone, outside the context of scenarios, is that they do not focus the analysis on parts of the system that are likely to be modified in the future [23]. If the values of the metrics suggest that a component is hard to modify, it is not always clear which kind of changes will be difficult in practice. Based on their experience with the tools, Gorton and Zhu also

emphasize the need for flexible tools that allow the user to define and customize views. The usefulness of the recovered views in an evaluation involving different kinds of system stakeholders is not explicitly considered.

6.2 Architecture Recovery

The DiscoTect approach [45] was used as a starting point in this project for the recognition of architectural elements from data gathered at runtime. However, DiscoTect's recognition approach based on Colored Petri Nets was found to be problematic when parts of recognized components indirectly call each other (section 3.3.2) and therefore an approach based on Prolog was used instead. The DiscoTect approach was used by Ganesan et al. [15] in an industrial case study. They recover architectural views in UML notation to check whether a system complies to its as-designed architecture. Although they use different tools than those proposed in [45], their approach is based on Colored Petri Nets. Indirect method calls between components are not discussed.

Queries on the Abstract Syntax Tree (AST) obtained by parsing the source code of the system under analysis are used in [20] to recognize instances of elements of architectural styles. Coverage metrics are proposed, which can be used to estimate how much of the system is understood. Yeh et al. [51] discuss how to visualize the recognized elements in views and how views can be combined to create additional views. Aside from the notation, the views presented in chapter 4 resemble the ones in [51]. The main difference is that Yeh et al. only use static analysis, as a result of which it may be harder to limit the scope of the recovery effort to the parts of the system that are relevant for a scenario-based architecture evaluation. This is also the case in [28], which proposes the use of Prolog queries as part of an approach to identify and visualize architectural concepts.

Guo et al. [18] store data extracted with static and dynamic analysis in a relational database and use SQL queries to recognize pattern instances. A disadvantage of SQL queries over Prolog queries is that they are often more difficult to implement and can lead to rulesets which are harder to maintain. For example, the order in which the queries are run is important and must be specified manually. However, the proposed recovery toolset is flexible, allowing other recognition tools to be integrated into it.

Richner and Ducasse [41] use Prolog rules to recover architectural views based on static and dynamic analysis. The recovered views show elements (such as components consisting of a group of classes) and relations (such as method calls) between them. The views recovered in chapter 4 somewhat resemble their views, although they do not use UML and do not recognize components based on observed behavior. Classes are mostly grouped into components manually, based on their class category (Smalltalk).

Static and dynamic analysis are combined in [42] to create two views: a graph representing the static structure of the system and a sequence chart showing the interaction between the elements in the structural view. The views are kept synchronized, expanding (collapsing) nodes in the structural view also reveals (hides) the interactions between the members of the composite node in the sequence chart. Prolog rules are used to specify how source code elements are grouped into more abstract elements and to specify which elements and

relations are to be shown in the views. The grouping is specified manually, as opposed to specifying patterns of structure and interaction to be recognized automatically.

Haqqie and Shahid [19] use Prolog to recover architectural views in terms of automatically recognized design pattern instances and the interactions between them. Their goal is to recover the rationale for the use of the identified patterns and to determine the quality attributes achieved by them. They describe similar views as those recovered in chapter 4, but do not discuss an evaluation of their approach in practice. Furthermore, they only use static analysis and do not discuss ways to limit the scope of the analysis to the relevant parts of the system.

Bauer and Trifu [3] use Prolog to recognize instances of design patterns, which they use as *clues* to guide the automatic clustering of source code elements into architectural elements. This way, they intend to combine the strengths of the two approaches, resulting in a complete mapping of the source code to architectural elements (clustering) and recovered architectural elements that make sense to the system's stakeholders (pattern matching). Whether the latter is achieved in practice was not evaluated with a user study.

6.3 Pattern Matching

The use of Prolog to recognize instances of design patterns in a software system has been proposed by several authors, e.g. [27, 3]. A disadvantage of using Prolog is that it is relatively hard to recognize instances which slightly deviate from the "textbook" versions of patterns. Several other approaches for design pattern detection have been proposed [12], including approaches which perform approximate matching. Evaluating the use of such approaches to recover architectural views for an ATAM evaluation would be interesting future work.

Another interesting approach is the recognition of patterns based on high-level specifications. For example, [50] discusses the transformation of UML sequence diagrams which specify the behavior of design patterns into finite automata which can be used to automatically recognize instances of the patterns. Such approaches are useful, because manually writing pattern specifications in Prolog is not very intuitive.

Most publications on design pattern detection focus on the number of (correctly) identified pattern instances. If visualization of the detected patterns is discussed at all, it does not focus on the usefulness of the visualizations for an ATAM evaluation.

6.4 Usability Evaluation

Most architecture recovery techniques and tools proposed in literature are accompanied with at least one case study to evaluate their use in practice. Reports on feedback from the system's stakeholders and the lessons learned in such case studies are often anecdotal. However, some empirical user studies have been done to evaluate the use of reverse engineering tools and techniques and the visualizations generated by them [10].

Cornelissen et al. [8] perform a controlled experiment to assess the usefulness of a trace visualization tool, measured in terms of the time needed for subjects to complete typical

program comprehension tasks and the correctness of their answers. They focus on program comprehension at a lower level of abstraction than the views recovered in chapter 4 and only include subjects who have at least some background in software development.

Knodel et al. [26] report significant differences in the effectiveness of two groups of subjects in performing architecture analysis tasks, where both groups analyze the same system, but are given different visualizations. Both visualizations were obtained through static analysis of the system, rather than using dynamic analysis to focus the analysis on a particular scenario. Furthermore, all subjects had a background in software architecture.

Driven by the requirements for an ATAM evaluation, the survey discussed in chapter 5 intentionally includes people with and without a strong background in software development. It focuses on whether a recovered architectural view covers the right parts of the system, at the right level of abstraction, in a way that can be understood by a wide range of stakeholders.

The understandability of UML diagrams has also been evaluated empirically. For example, Ricca et al. [40] performed experiments to evaluate the effect of the use of domain-specific UML stereotypes on the ability of the subjects to understand a software system. They found that the extent to which the use of stereotypes influenced program comprehension was dependent on the background of the subjects, in terms of abilities and experience. Since ATAM evaluations typically involve participants with diverse backgrounds, the results of such experiments can enable more adequate recovery of views that address the needs of all participants in an evaluation.

In many published studies, the subjects directly interact with a tool to evaluate its usability. The survey in chapter 5 leaves this as future work.

Chapter 7

Conclusions and Future Work

The goal of this master's project was to set up and validate a repeatable, tool-supported process for the recovery of architectural views for use in an ATAM evaluation. Based on a study of the available literature, the choice was made to use the Symphony architecture recovery process, to use execution tracing to obtain information about the system and to use a rule-based pattern matching approach to recover UML diagrams from the traces. This led to three main research questions (section 1.2), which will be answered below.

RQ1 *Which tools can be used to support and (as much as possible) automate architecture recovery using dynamic analysis and rule-based abstraction techniques?*

To answer this question, several tools were examined. No usable off-the-shelf execution tracer could be found, so a simple tracer was developed from scratch. An attempt was made to use DiscoTect to recover UML models from execution traces, but several problems were encountered with this tool. The prototype implementation had several bugs and poor performance. Worse, call dependencies between the recovered components could not always be identified correctly. As a result, the choice was made to use rules written in Prolog instead. Two additional tools were developed to help generate UML models in XMI format. Enterprise Architect was used (without modification) to visualize the recovered UML models.

To validate the recovery process and the supporting tools, a case study was performed in which an attempt was made to recover the views of the Exact Connectivity Layer found most useful by the architect when creating the architecture. Unfortunately, due to problems encountered when importing these diagrams into Enterprise Architect, this attempt failed early on.

However, the case study did show that the recovery process and tools can be used to recover UML component diagrams using dynamic analysis. Much of the work involved in recovering the diagrams was automated, but developing the ruleset and rearranging the generated views still had to be done manually. One of the main benefits of using dynamic analysis, the fact that it allows focusing only on the parts of the system involved in a particular scenario, is illustrated by the differences between the views obtained for the demo-document and word-document scenarios. However, sometimes workarounds were needed

because the execution trace did not contain all necessary information. Furthermore, a potential scalability issue was found: processing large traces may take too much time to allow an interactive and iterative process.

Based on these results, a preliminary answer to research question 1 can be given. The tools presented in chapter 3 appear to be able to support the recovery process and reverse engineering techniques outlined in chapter 2. The approach appears to have the potential to meet its goals, but more work is needed before the tools can be used in practice and a definitive answer can be given to this research question.

RQ2 *Can the tool-supported architecture recovery process be used to validate existing architectural documentation?*

Unfortunately, validation of existing "hand-drawn" architectural views was not possible, because the kinds of diagrams that were available could not be imported correctly into Enterprise Architect, making the recovery of these diagrams impossible with the available tools. However, the case study did show that several design decisions (that could be relevant in an ATAM evaluation) could be validated against the recovered as-built architecture. Although the results of the preliminary case study are hard to generalize to other views, systems, scenarios and design decisions, the results are promising. It is quite possible that, with more work, the answer to this question can be "yes".

RQ3 *Can the tool-supported architecture recovery process be used to recover documentation that is considered readable and useful in practice by the system's stakeholders?*

To answer this question, a survey was conducted (chapter 5). The results varied from stakeholder to stakeholder, but overall the stakeholders appeared to be mildly positive about the readability and usefulness of the recovered view. The view was certainly not considered useless by a majority of respondents, in fact some respondents suggested additional applications for the view. Because only a preliminary investigation was performed, more work is needed to provide a more conclusive answer to research question 3. Several directions on which future work could be focused were identified, including possible ways in which the recovered view could be improved. In particular, the recovery of different abstractions should be looked at, to improve the scalability of the view and to reduce the amount of text accompanying the view that, for now, has to be written completely manually, which reduces the level of automation, reusability and repeatability.

Overall Results Overall, the results indicate that the architecture recovery approach and tools proposed in this thesis have the potential to be useful for recovering views for use in an ATAM evaluation. Further development of the tools will be necessary before they can be used in practice on a regular basis. Further evaluation of the usefulness of the recovery process, tools and resulting views is also necessary. Given the potential benefits of supporting architecture evaluation with architecture recovery techniques and given that initial experiences were mildly positive, further development appears to be worthwhile.

7.1 Lessons Learned and Future Work

Several lessons have been learned throughout the project, indicating ways in which the architecture recovery approach could be improved. Furthermore, several questions have remained open. This section gives an overview of these lessons, questions and possible improvements and the main directions for future work to address them.

7.1.1 Data Gathering

In practice, a combination of static and dynamic analysis is needed to adequately detect patterns in software systems [21]. In the case study for instance, static analysis could have enabled automatic recognition of the generic entity service. Because this was not feasible in the context of this project, it is left as future work. To enable a goal-driven approach, data obtained with dynamic analysis can be used to slice the data obtained with static analysis, so that only elements involved in the execution of a particular scenario are shown [49].

The overhead introduced by the tracer caused timeouts. Working around this problem may not always be possible or desirable, therefore ways to reduce tracing overhead should be investigated. This could include optimizing the tracer, or trying other tracing approaches which may have better performance. Another option is to limit the parts of the system that are traced to the minimum needed to recognize architectural concepts. For example, Heuzeroth et al. [21] perform static analysis to identify parts of a program which potentially implement a design pattern. These parts are then instrumented, so that dynamic analysis can be used to remove false positives from the set of potential pattern instances.

It is not always clear how to translate ATAM evaluation scenarios into execution scenarios, in particular *exploratory scenarios* may cause problems. These scenarios propose large changes in the requirements to the system, intended to "stress" the architecture [24]. As a result it might be hard to find representative execution scenarios that result in traces that actually support the evaluation.

Manually running the scenarios is tedious work. If they are run automatically, for instance by using testing tools, a larger numbers of scenarios could be traced. If changes to the architecture do not require major changes to the rulesets, this would support automatic updating of architectural views for new versions of a system and enable frequent architecture conformance checks.

7.1.2 Abstraction

Traces could be combined to allow a view to cover a larger part of the system. For example, by merging the traces in the case study with a trace in which the generic entity service is actually used as a webservice, it may be possible to automatically recognize the generic entity webservice. Traces of the clients could also be included, allowing a single view to show parts of both the clients and the server.[1] Furthermore, including more than one code path may increase the probability of finding architectural violations.

[1] Observing the behavior of clients may require different tracing techniques, because they may be based on different platforms.

One way in which traces might be combined is to split the recognition rules into multiple parts, with low-level rules operating on individual traces and high-level rules combining facts derived from multiple traces by the low-level rules. Furthermore, methods that are involved in (almost) all traces could be identified, as proposed in [17]. This might allow automatic recognition of the entity engine, which had to be mapped manually in the case study.

To reduce the startup effort involved in analyzing a new system, a library of rules which can recognize the most common architectural concepts should be created. More work is also needed to improve the reusability of the (library) rulesets. At the moment it is difficult to write rules which sufficiently cover the possible ways in which an architectural concept can be implemented. Including approaches proposed in literature for the approximate matching of design patterns might be needed to address this issue.

The response from the participants in the survey indicates that higher-level abstractions, such as layers, may need to be recognized. At the moment, unless the parts of a layer match some pattern, parts can only be assigned to layers manually. Future work could include finding such patterns or looking for other techniques to group parts into layers automatically.

7.1.3 Presentation

Currently, it is necessary to manually write code which serializes model elements in XMI format. This requires detailed knowledge of the UML metamodel and distracts the analyst from the actual architecture recovery task. Providing a more high-level way to specify modifications to the recovered architectural view would be very helpful.

For instance, the analyst could model templates of architectural elements in a tool such as Enterprise Architect. These are then exported in XMI format, based on which XMI serialization code is generated. Ideally, in combination with specifying the structure and behavior of the patterns to recognize in UML (similar to what is done in [50]), this would allow an analyst to generate all architecture recovery code, by modeling what should be recognized and how the recognized elements should be represented in a view. This also makes it easier for individual analysts to customize the visualization according to their preferences. Based on the results of their experiment, Knodel et al. argue that such configurability should be a requirement for visualization tools [26].

One of the survey participants suggested allowing the user to zoom in to a more detailed architectural view. Future work should investigate whether this does not make the recognition, presentation and view generation rules unmanageably complex. The use of colors to distinguish different types of elements was also suggested. This should be possible, because UML allows alternative symbols to be used to represent stereotyped elements and Enterprise Architect appears to support this.

7.1.4 Evaluation

Future work on evaluating the recovery approach includes using the recovered views in an actual ATAM evaluation. This could more conclusively show whether the recovered views are at the right level of abstraction and contain the information necessary for an ATAM.

Only structural views have been recovered in the case study, primarily containing element types identified by interviewing the architect. It remains to be seen whether this approach allows recovery of other views, possibly containing abstractions of concern primarily to other stakeholders.

Several things that are important for the success of this approach in practice have not been investigated in the case study and survey:

- The perceived difficulty of creating rulesets.
- Reuse of rulesets on scenarios that deviate more significantly from each other than the demo-document and word-document scenarios.
- Do different analysts given the same task obtain similar results? This is an important aspect of repeatability in practice.

Finally, experiences with the combined recovery-evaluation approach need to be compared against those with other approaches published in literature, such as SQUA^{3}RE.

Bibliography

[1] Muhammad Ali Babar and Ian Gorton. Comparison of scenario-based software architecture evaluation methods. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 600–607. IEEE Computer Society, 2004.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, second edition, 2003. ISBN 0-321-15495-9.

[3] Markus Bauer and Mircea Trifu. Architecture-aware adaptive clustering of OO systems. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 3–14. IEEE Computer Society, 2004.

[4] Alan Bryman. *Social Research Methods*. Oxford University Press, third edition, 2008. ISBN 978-0-19-920295-9.

[5] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Automated clustering to support the reflexion method. *Information and Software Technology*, 49:255–274, March 2007.

[6] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. SEI Series in Software Engineering. Addison-Wesley, 2001. ISBN 0-201-70482-X.

[7] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[8] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC'09)*, pages 100–109. IEEE Computer Society, 2009.

[9] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *WICSA '04:*

Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, pages 122–132. IEEE Computer Society, 2004.

[10] Massimiliano Di Penta, R. E. K. Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 281–285. IEEE Computer Society, 2007.

[11] Liliana Dobrica and Eila Niemelä. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.

[12] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(6):823–855, 2009.

[13] Ecma International. ECMA-335: Common Language Infrastructure (CLI), Fourth Edition. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, 2006.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[15] Dharmalingam Ganesan, Thorsten Keuler, and Yutaro Nishimura. Architecture compliance checking at run-time. *Information and Software Technology*, 51(11):1586–1600, 2009.

[16] Ian Gorton and Liming Zhu. Tool support for just-in-time architecture reconstruction and evaluation: An experience report. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 514–523. ACM, 2005.

[17] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society, 2005.

[18] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A software architecture reconstruction method. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture*, pages 15–34. Kluwer, B.V., 1999.

[19] Sarah Haqqie and Arshad Ali Shahid. Mining design patterns for architecture reconstruction using an expert system. In *9th International Multitopic Conference, IEEE INMIC 2005*, 2005.

[20] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 186–195. ACM, 1995.

[21] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 94–103. IEEE Computer Society, 2003.

[22] Kurt Jensen. An introduction to the theoretical aspects of coloured petri nets. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency, Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 230–272. Springer Berlin / Heidelberg, 1994.

[23] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, 1996.

[24] Rick Kazman, Len Bass, Mark Klein, Tony Lattanze, and Linda Northrop. A basis for analyzing software architecture analysis methods. *Software Quality Journal*, 13(4):329–355, 2005.

[25] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[26] Jens Knodel, Dirk Muthig, and Matthias Naab. An experiment on the role of graphical elements in architecture visualization. *Empirical Software Engineering*, 13:693–726, December 2008.

[27] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, pages 208–215. IEEE Computer Society, 1996.

[28] Nabor C. Mendonça and Jeff Kramer. Developing an approach for the recovery of distributed software architectures. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, pages 28–36. IEEE Computer Society, 1998.

[29] Paul Metselaar. Repeatable methods for software architecture recovery and evaluation – literature study, 2009.

[30] Metadata (unmanaged api reference). `http://msdn.microsoft.com/en-us/library/ms404384%28v=VS.90%29.aspx`.

[31] Profiling (unmanaged api reference). `http://msdn.microsoft.com/en-us/library/ms404386%28v=VS.90%29.aspx`.

[32] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[33] Object Management Group. Diagram interchange, version 1.0. `http://www.omg.org/spec/UMLDI/1.0/`, 2006. OMG Document Number: formal/06-04-04.

[34] Object Management Group. Meta object facility (MOF) core specification, version 2.0. http://www.omg.org/spec/MOF/2.0/, 2006. OMG Document Number: formal/06-01-01.

[35] Object Management Group. MOF 2.0/XMI mapping, version 2.1.1. http://www.omg.org/spec/XMI/2.1/PDF, 2007. OMG Document Number: formal/2007-12-01.

[36] Object Management Group. OMG unified modeling language (OMG UML), infrastructure, v2.1.2. http://www.omg.org/spec/UML/2.1.2/, 2007. OMG Document Number: formal/2007-11-04.

[37] Object Management Group. OMG unified modeling language (OMG UML), superstructure, v2.1.2. http://www.omg.org/spec/UML/2.1.2/, 2007. OMG Document Number: formal/2007-11-02.

[38] Krisztián Pócza, Mihály Biczó, and Zoltán Porkoláb. Towards effective runtime trace generation techniques in the .NET framework. In *Short Communication Papers Proceedings of the 4th .NET Technologies Conference*, pages 9–16, 2006. Available at http://dotnet.zcu.cz/NET_2006/NET_2006.htm.

[39] Damien Pollet, Stéphane Ducasse, Loïc Poyet, Ilham Alloui, Sorana Cîmpan, and Hervé Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 137–148. IEEE Computer Society, 2007.

[40] Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, and Mariano Ceccato. The role of experience and ability in comprehension tasks supported by uml stereotypes. In *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, pages 375–384. IEEE Computer Society, 2007.

[41] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 13–22. IEEE Computer Society, 1999.

[42] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. In *CSMR '02: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 47–55. IEEE Computer Society, 2002.

[43] Banani Roy and T. C. Nicholas Graham. An iterative framework for software architecture recovery: An experience report. In *ECSA '08: Proceedings of the 2nd European conference on Software Architecture*, pages 210–224. Springer-Verlag, 2008.

[44] Banani Roy and T.C. Nicholas Graham. Methods for evaluating software architecture: A survey. Technical Report 2008-545, Queen's University School of Computing, 2008. http://research.cs.queensu.ca/TechReports/Reports/2008-545.pdf.

[45] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[46] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-oriented visualization. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 389–405. ACM, 1996.

[47] Christoph Störmer. *Software Quality Attribute Analysis by Architecture Reconstruction (SQUA^{3}RE)*. PhD thesis, Vrije Universiteit Amsterdam, March 2007.

[48] Terrance Swift, David S. Warren, Konstantinos Sagonas, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Diptikalyan Saha, Steve Dawson, and Michael Kifer. *The XSB System Version 3.2, Volume 1: Programmer's Manual*. `http://xsb.sourceforge.net/`.

[49] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 304–313. IEEE Computer Society, 1999.

[50] Lothar Wendehals and Alessandro Orso. Recognizing behavioral patterns at runtime using finite automata. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 33–40. ACM, 2006.

[51] Alexander S. Yeh, David R. Harris, and Melissa P. Chase. Manipulating recovered software architecture views. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 184–194. ACM, 1997.

[52] Robert K. Yin. *Case Study Research, Design and Methods*. SAGE Publications, Inc, fourth edition, 2009. ISBN 978-1-4129-6099-1.

[53] Andy Zaidman. *Scalability Solutions for Program Comprehension Through Dynamic Analysis*. PhD thesis, University of Antwerp, 2006.

Appendix A

Glossary

ADL Architecture Description Language

ATAM Architecture Tradeoff Analysis Method [2]

CLR Common Language Runtime

CRUD Create, Retrieve, Update, Delete

DiscoSTEP Discovering Structure Through Event Processing [45]

EA Enterprise Architect

JIT Just-In-Time

MOF Meta Object Facility [34]

SOAP Simple Object Access Protocol

SQL Structured Query Language

UML Unified Modeling Language [37]

XMI XML Metadata Interchange [35]

XML Extensible Markup Language

Appendix B

Trace Formats

This appendix contains a description of the formats of the trace files written by the TraceProcessor output plugins.

B.1 XML

The TraceProcessor output plugin for DiscoTect writes method call events in an XML format, following the XML schema shown below. An example call event is shown in figure 3.6.

```
<xs:element name="call">
  <xs:complexType>
    <xs:sequence>
      <!-- names, types and values of method parameters -->
      <xs:element name="arg" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required" />
          <xs:attribute name="type" type="xs:string" use="required" />
          <xs:attribute name="value" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>

    <!-- the ID of the object on which the method was invoked (if any) -->
    <xs:attribute name="calleeID" type="xs:string" use="optional" />
    <!-- namespace and name of the class of which the object is an instance -->
    <xs:attribute name="calleeNS" type="xs:string" use="optional" />
    <xs:attribute name="calleeType" type="xs:string" use="optional" />

    <!-- namespace and name of the class containing the method's implementation.
         may differ from calleeNS/calleeType if the method is inherited or static. -->
    <xs:attribute name="calleeOwnerNS" type="xs:string" use="required" />
    <xs:attribute name="calleeOwnerType" type="xs:string" use="required" />

    <!-- sequence number of the call -->
    <xs:attribute name="timestamp" type="xs:string" use="required" />
    <!-- visibility of the method (public, private, etc...) -->
    <xs:attribute name="visibility" type="xs:string" use="required" />
```

```
    <!-- true if the method is static, false otherwise -->
    <xs:attribute name="static" type="xs:boolean" use="required" />
    <!-- true if the method is public, false otherwise -->
    <xs:attribute name="constructor" type="xs:boolean" use="required" />
    <!-- name of the called method -->
    <xs:attribute name="method" type="xs:string" use="required" />
    <!-- type of the return value -->
    <xs:attribute name="returnType" type="xs:string" use="required" />
    <!-- ID of the object from which the call was made, if any -->
    <xs:attribute name="callerID" type="xs:string" use="optional" />
    <!-- sequence number of the call to the caller of this method, if any -->
    <xs:attribute name="callerTimestamp" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
```

This output format is highly redundant, for example, the name of a method is included each time it is called. An alternative is to only include a method ID and generate an "event" for each method containing its ID, name, etc... However, the DiscoSTEP rules then have to combine a "method event" with each call event, which significantly reduces DiscoTect's performance for large traces.

Return values are not included in this schema, but could easily be added.

B.2 Prolog

The Prolog output plugin generates a plaintext file containing Prolog facts. Table B.1 gives an overview of the kinds of facts that are included. Figure 3.9 shows a part of an actual set of facts.

Fact	Description

`className(Class, Name)`
`methodName(Method, Name)`
`namespaceName(Method, Name)`
`assemblyName(Method, Name)`
	Defines the names of runtime entities.

`fullTypeName(Method, Name)`	Similar to methodName, but also includes type arguments of instances of generic types.
`superClass(Subclass, Superclass)`	Represents inheritance. Does not include implemented interfaces.

`classMember(Method, Class)`
`namespaceMember(Class, Namespace)`
`assemblyMember(Class, Assembly)`
	Defines containment of methods in classes and classes in namespaces and assemblies.

`parameters(Method, List)`	List contains a list of lists, each containing the name and type of a method parameter, in left-to-right order.
`returnType(Method, Type)`	Return type of a method.
`instanceof(Object, Class)`	Object is an instance of Class.

`methodCall(SequenceNumber, CallerObject,`
`CallerMethod, CalleeObject,`
`CalleeMethod, Previous)`
	CallerMethod of CallerObject calls CalleeMethod of CalleeObject. Each call has a unique sequence number. Previous is the sequence number of the call to the caller.

`parameterValues(SequenceNumber, List)`	List of string representations of the parameter values passed in a method call, in left-to-right order.
`returnValue(SequenceNumber, Value)`	String representation of the value returned by a method, if any.
`marker(SequenceNumber, Id, Description)`	Generated for each marker inserted into the trace.
`constructor(Method)`	Generated if Method is a constructor. Similar facts are generated to indicate whether a method is static, public, private, etc...

Table B.1: Facts generated by the TraceProcessor Prolog output plugin.

Appendix C

View Evaluation Questionnaire

This appendix contains the questionnaire used in the survey discussed in chapter 5. This includes the introduction, the recovered view (representing the demo-document scenario in iteration 6 of section 4.2) and the accompanying description.

Architectural View Evaluation

Introduction

The architecture of a software system has a large influence on the quality attributes of the system, such as its performance, security and interoperability. By evaluating the architecture, we can identify the architectural design decisions that influence these quality attributes and look for ways to improve the system. Furthermore, the lessons learned from past evaluations can be applied to new systems right from the start.

To evaluate an architecture, we need an accurate description of it. However, the architectural documentation and the source code of a system tend to "drift apart" over time, because if one is changed, the other is not updated automatically. This could lead to invalid evaluation results. For example, based on its documentation a system may appear to be easy to modify, even if it is hard to modify in practice.

To address this issue, it is necessary to check that the architectural documentation matches the actual system, and to recover any unavailable documentation that might be needed for an evaluation. A set of tools has been set up to support and partially automate this.

The goal of this survey is to assess whether the architectural views recovered with these tools are readable and usable in practice. First, a recovered UML component diagram is shown, then a number of questions are asked about the diagram, to assess the extent to which it is useful for you. Note that the questions have no right and no wrong answers. Your personal opinion is what is important.

Context

To put your answers to the questions in context, please rate your level of knowledge of, or experience with, the following topics, on a scale of 1 (no knowledge/experience) to 5 (in-depth knowledge/expert):

	1	2	3	4	5
1. UML Component Diagrams	☐	☐	☐	☐	☐
2. functional requirements of the Connectivity Layer	○	○	○	○	○
3. non-functional requirements of the Connectivity Layer (such as performance, modifiability, …)	☐	☐	☐	☐	☐
4. architecture of the Connectivity Layer	○	○	○	○	○
5. technical design of one or more parts of the Connectivity Layer	☐	☐	☐	☐	☐
6. implementation of one or more parts of the Connectivity Layer	○	○	○	○	○

Recovered view

On the following page, a UML component diagram recovered from the Exact Connectivity Layer is shown. This diagram was recovered by logging the method calls that occurred at the server side as a result of performing the following usage scenario:

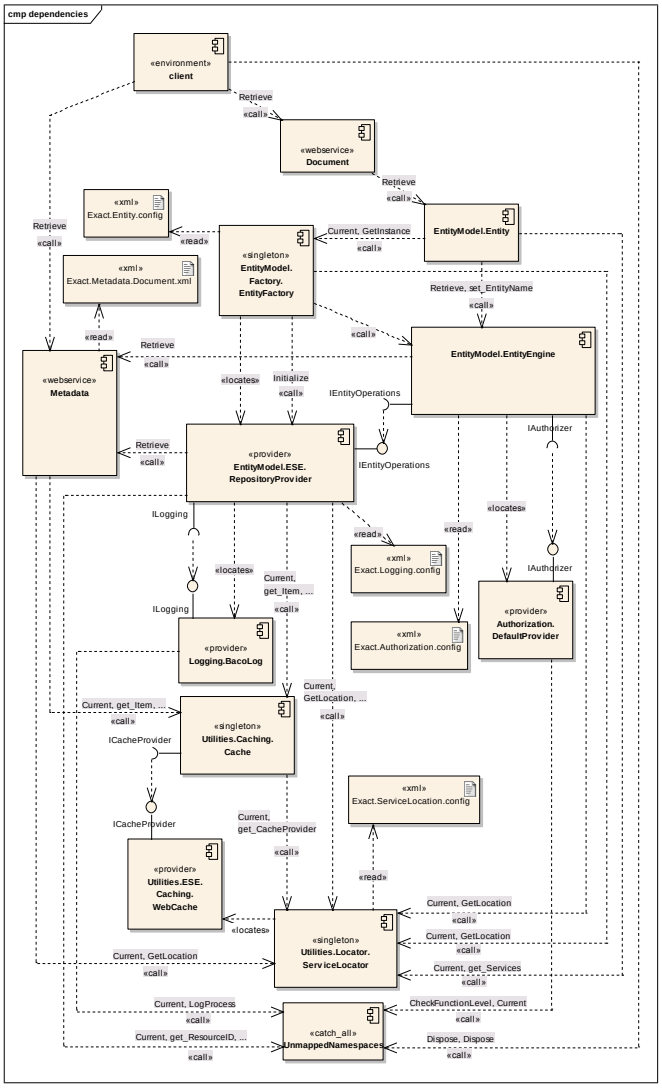
1. The web server running Synergy and the Connectivity Layer was reset
2. Using the Connectivity Demo Application, the metadata for the Document entity was retrieved
3. Again using the Connectivity Demo Application, information was retrieved about a document stored in Synergy

A set of rules was then applied to the logged method calls to automatically recognize the components and relations involved in processing these actions. The recognized components and relations are shown in the component diagram.

The client (the demo application) is represented by the component at the top of the diagram. It uses the *Retrieve* method of the *Metadata* and *Document* webservices to retrieve the requested data. The *providers* used by these webservices to handle the requests are also shown. Furthermore, several kinds of dependencies between the components are shown.

At the bottom of the diagram a component called “UnmappedNamespaces” is shown. This component contains all code involved in handling this scenario that is not part of the Connectivity Layer. This includes all parts of Synergy Enterprise involved in this scenario. This allows us to see the dependencies between the Connectivity Layer and the underlying system. Clearly, all communication between the Connectivity Layer and Synergy is performed via providers, as designed. It is also clear that the Metadata service does not communicate with Synergy at all. It obtains all its information from its own XML files. This also matches the original design. Since the Connectivity Layer should present the same view of an entity to the “outside world”, regardless of the underlying system, the Metadata service should not directly depend on any particular underlying system. This strong separation between the Connectivity Layer and the underlying system make it very flexible.

Because the actual implementation closely matches the design, it is very likely that the system is not only flexible in theory, but also in practice.


cmp dependencies
«environment» client
Retrieve «call»
«webservice» Document
Retrieve «call»
«xml» Exact.Entity.config
Retrieve «call»
«read»
«singleton» EntityModel. Factory. EntityFactory
Current, GetInstance «call»
EntityModel.Entity
«xml» Exact.Metadata.Document.xml
Retrieve, set_EntityName «call»
«read»
«call»
EntityModel.EntityEngine
Retrieve «call»
«webservice» Metadata
«locates»
Initialize «call»
IEntityOperations
IAuthorizer
Retrieve «call»
«provider» EntityModel.ESE. RepositoryProvider
IEntityOperations
ILogging
«locates»
«read»
«read»
«locates»
«xml» Exact.Logging.config
IAuthorizer
Current, get_Item, ... «call»
ILogging
«provider» Logging.BacoLog
«xml» Exact.Authorization.config
«provider» Authorization. DefaultProvider
Current, get_Item, ... «call»
Current, GetLocation, ... «call»
ICacheProvider
«singleton» Utilities.Caching. Cache
«xml» Exact.ServiceLocation.config
ICacheProvider
Current, get_CacheProvider «call»
«read»
«provider» Utilities.ESE. Caching. WebCache
«locates»
Current, GetLocation «call»
Current, GetLocation «call»
«singleton» Utilities.Locator. ServiceLocator
Current, get_Services «call»
Current, GetLocation «call»
Current, LogProcess «call»
CheckFunctionLevel, Current «call»
Current, get_ResourceID, ... «call»
«catch_all» UnmappedNamespaces
Dispose, Dispose «call»

Evaluation

1. Please rate the extent to which you agree with the following statements, on a scale from 1 to 5 (1=strongly disagree, 2=disagree, 3=neither agree nor disagree, 4=agree, 5=strongly agree):

	1	2	3	4	5
a. I recognize the components and relations shown in the view	☐	☐	☐	☐	☐
b. the elements and relations shown in the view correspond to concepts I normally use to reason about the system	○	○	○	○	○
c. the view contains too many elements and relations	☐	☐	☐	☐	☐
d. the description accompanying the view made it easier to understand the view	○	○	○	○	○
e. the view gives me new insights into the system	☐	☐	☐	☐	☐

2. Is the view correct?

- ○ yes
- ○ no
- ○ cannot determine

If the view is incorrect, please describe what should be changed to make the view correct.

3. Please indicate whether you think this view could be useful for the activities listed below. Please rate each activity on a scale from 1 (not useful at all) to 5 (very useful).

	1	2	3	4	5	no opinion
a. adding a new feature	☐	☐	☐	☐	☐	☐
b. fixing a bug	○	○	○	○	○	○
c. estimating the amount of time or effort needed for a task	☐	☐	☐	☐	☐	☐
d. finding ways to improve the quality of the system (such as performance, or modifiability)	○	○	○	○	○	○

If this view could be useful for activities not listed above, please add them here:

4. Does the view show all elements and relations you consider essential?

- ○ yes
- ○ no

If essential elements or relations are missing, please describe what you think should be added (or refined further) to make the view complete

5. Does the view contain elements or relations that you think are irrelevant?

- ○ yes
- ○ no

If the view contains irrelevant elements or relations, which ones do you think should be removed, or grouped into more abstract elements or relations?

6. Please rate the extent to which you agree with the following statements, on a scale from 1 to 5 (1=strongly disagree, 2=disagree, 3=neither agree nor disagree, 4=agree, 5=strongly agree):

	1	2	3	4	5
a. this view is useful for a novice member joining my team	☐	☐	☐	☐	☐
b. overall, this view is useful	○	○	○	○	○

7. If you have any further comments or remarks, please feel free to write them down below.

Thanks!

Thank you for participating in this survey. Please save the completed survey form (as .docx or .pdf) and e-mail it to Paul Metselaar. The results of the survey will be published in Synergy. If you would like to be notified of this by e-mail, please indicate this in your e-mail.

Thanks!

Appendix D

View Evaluation Survey Results

This appendix contains the results of the questionnaire. To protect the privacy of the participants, the participants are listed in random order and their names are not included.

D.1 Context

Table D.1 lists the answers given to the context identification questions. The referenced stakeholder groups are listed in table D.2.

Question \ Participant	1	2	3	4	5	6	7	8
Stakeholder group	P	O	P	D	R	A	D	A
1 (component diagrams)	2	4	2	4	4	4	2	3
2 (functional requirements)	5	3	3	4	2	4	3	3
3 (non-functional requirements)	4	2	2	4	2	5	3	2
4 (architecture)	3	3	2	4	2	5	3	3
5 (technical design)	5	2	1	4	2	5	3	2
6 (implementation)	5	1	2	4	2	4	3	3

Table D.1: Answers to the context identification questions.

Abbreviation	Stakeholder group
A	Software architects team
D	Development
O	Exact Online
P	Product management
R	Research team

Table D.2: Stakeholder groups.

Question \ Participant	1	2	3	4	5	6	7	8
1a	4	3	3	4	5	5	4	3
1b	2	4	2	3	4	4	3	3
1c	4	2	4	4	3	3	4	3
1d	5	4	2	4	3	4	4	4
1e	4	4	4	4	5	4	3	3
2 (view correct)	yes	cd	cd	yes	cd	no	yes	cd
3a (feature addition)	5	4	2	3	5	5	2	4
3b (bug fixing)	5	2	5	2	5	4	2	5
3c (effort estimation)	5	2	2	3	5	3	2	4
3d (quality improvement)	5	4	1	3	5	4	3	3

Table D.3: Answers to questions 1, 2 and 3.

D.2 Evaluation

Table D.3 shows the answers to questions 1, 2 and 3. In the answers to question 2, "cd" means "cannot determine". Participant 6 noted that the view was incorrect because the "main authorization component which actually reads authorization config and calls specific authorization provider" was missing.

In response to question 3, three participants mentioned additional activities for which the view could be useful:

- *Participant 1:* "For documentation. Preferable dynamical generated."
- *Participant 2:* "As architecture for similar kind of solutions (think web services for Exact Online)"
- *Participant 4:* "Assessing Security / Threat analysis"

Except for participant 2 and 3, all participants answered "yes" to question 4. Participant 2 answered "no", commenting "Security & data access. Database access. I miss a bit the layers (tiers) within the architecture". Participant 3 commented "This I could not answer, but I can imagine that db access or business rules application are also relevant for certain cases."

Five participants answered "no" to question 5. The comments of the participants who answered "yes" are listed below:

- *Participant 1:* "`Utilities.Locator.FacilityLocator`"
- *Participant 4:* "Use `<<call>>`will be sufficient enough instead of showing the method like `Current`, `GetLocation`, `GetInstance` and etc. This will make the view cleaner and not confusing the novice member."
- *Participant 5:* "I wonder if the config files should be part of this view. I would consider moving the config usage to a lower level."

Question \ Participant	1	2	3	4	5	6	7	8
6a (new team member)	4	4	4	2	4	5	3	2
6b (overall use)	3	4	4	4	5	4	3	3

Table D.4: Answers to question 6.

Table D.4 lists the answers given to question 6. Question 7 asked the participants to write down any further comments or remarks they might have, which several participants did:

- *Participant 2:* "I only see a few interfaces defined. I would expect all main interfaces for the components to be defined. Use of colors to distinct components of a given type (assemblies, xml etc) Can I also zoom into a more detailed architectural view?"
- *Participant 3:* "I think it would be more useful to have made this for a not recent project. Now I always have the feeling that we know much more and for a given old architecture it might be so much more an eye-opener on what you can retrieve 'automatically'."
- *Participant 5:* "I would consider making two versions of this diagram, first one is this view, second one is a more abstract one, level out more details and only showing the essential parts (client, entrypoints to the outside world, entrypoints to other (underlying) components."
- *Participant 7:* "Quite hard to make judgment based on a single diagram, because it only layout the component relationship. In fact, more is requires for different tasks. For example, fixing bugs requires more details than function call relationship. Purpose and operation done in each component is equally important. New team members require more background information and overview diagram. (normally not drawn using UML)"
- *Participant 8:* "Perhaps new tools such as the Visual Studio 2010 Architecture Explorer could help in determining the degree of dependency of an individual component. It would help to gauge what would be the impact if refactoring or feature expansion could have for the solution from a high level perspective."