From Package to Process

Dynamic software architecture reconstruction using process mining

Tijmen de Jong

From Package to Process:
Dynamic software architecture reconstruction using process mining

Master Thesis Business Informatics

Tijmen de Jong

March 2019

First supervisor: Jan Martijn van der Werf

Second supervisor: Jurriaan Hage

Utrecht University

1 ABSTRACT

Despite the advances in software architecture, real-world software systems often don't have an up-to-date architecture available. This reduces the ability of stakeholders to reason about the system in question. For this reason, software architecture reconstruction is as relevant as ever.

Different software architecture reconstruction approaches give different perspectives of the system under study (SUS). We choose to develop an approach that uses dynamic inputs to create a hierarchical view that shows the interactions of different elements within the SUS. This hierarchical interaction model (HIM) features collapsible elements, where any higher-level hierarchical element can be collapsed to abstract its underlying structure and behavior.

To demonstrate our technique, we created a ProM plugin. With this plugin users can create a HIM from a log and select elements to extract their interactions. A process model can then be mined from the extracted interactions. We test the accuracy of our approach by comparing logged behavior of an example program with its code. We then demonstrate the HIM visualization using our ProM plugin. Next, we instrument the open source bibliography manager JabRef to generate a log of its behavior. We create a HIM visualization from JabRef and use a pre-made architecture mapping to compare the implemented architecture of JabRef to its intended architecture. Finally, we take small samples of the interactions and use them to mine process models from parts of JabRef's logged behavior.

Evaluating our results, we find that our approach creates accurate models without developer effort. Focusing on the dynamics of software, our approach has different pros and cons compared to static approaches. The main difference is caused by relying on the quality of runtime scenarios to capture the system's architecture, while static approaches rely on analyzing source code. In practice this means that during architecture conformance checking, our approach delivers different architecture violations. While viable on its own, our approach is complementary to a static architecture reconstruction approach.

CONTENTS

1 INTRODUCTION

Modern software systems are often complex, making it difficult for stakeholders to understand and reason about them. To aid in the understanding of a software system, software architectures are used [1]. When a software architecture is used prescriptively, the actual architecture of a system should follow the intended architecture. As time passes, the architecture of an implemented system often starts drifting away from its intended (documented) architecture [2]. This mismatch makes it more difficult to employ the architecture for its main uses: understanding, reuse, construction, evolution, analysis and management [3]. Alternatively, there could be no available architecture at all, making analysis of the implemented system even more difficult.

Because the mismatch between intended and actual architecture is a commonly occurring problem, the field of software architecture reconstruction (SAR) exists [2]. Ducasse and Pollet define SAR as "a reverse engineering approach that aims at reconstructing viable architectural views of a software application" [2]. They also describe several different inputs that can be used to reconstruct a software architecture from. The two most popular are static information (source code) and dynamic information (traces, logs, events, etc.).

The advantage of using static information is that it gives a complete overview of the software: all classes, methods, functions, etcetera can be analyzed. Conversely, dynamic analysis provides insight into the runtime behavior of a software system. It also allows the exposure of object identities and aspects that cannot be resolved at compile-time [4] as well as analysis of runtime polymorphism and interactivity within a system [5]. Many systems combine both static and dynamic information for (semi-)automatically building an architecture [2], [4], [6].

One of the sources of complexity in modern software systems is concurrency. Concurrent systems have become common and representing the concurrency of these systems in the architecture is often helpful [1, p. 333]. Despite this, most systems for reverse engineering dynamic approaches do not support multi-threaded or distributed systems [4].

Currently, few software architecture reconstruction approaches exist that use dynamic inputs to create a visualization of the run-time structure as well as the behavior of a system. Of the approaches that we found, few of the approaches took object-orientation and concurrency into account. If they did, they required intimate knowledge of the source code from the user.

1.1 OBJECTIVES

An effective SAR approach for visualizing the run-time structure and behavior of a system represents the object-oriented aspects of software, models concurrency accurately, models the behavior of the system from software execution data, doesn't require modification of the source code for instrumentation and doesn't require knowledge of the source code.

This does not mean that *only* approaches that meet these objectives are useful or even interesting. Approaches based on static or hybrid inputs have different advantages compared to those with dynamic inputs. Also, not every system has interesting or even any object-oriented or

concurrent aspects. These objectives were chosen to cover a large class of real-world systems, and assumes the user is interested in the run-time structure and behavior of the system. In the remainder of this section, we will go through each of the requirements and explain their importance.

1.1.1 Represents the object-oriented aspects of software

Object-oriented languages are commonly used in industry. Their use is widespread across all kinds of software systems. Many of the most popular programming languages are object-oriented[1]. A system that represents these aspects is therefore relevant to many real-world systems. The paradigm of object orientation significantly affects the more low-level aspects of software architecture: classes, objects and packages/namespaces are building blocks of more implementation-specific views. Information about these aspects is therefore useful in understanding these systems.

In addition to the popularity of object-orientation in industry, other programming paradigms are far less frequently researched in dynamic architecture reconstruction [4]. The exception would be procedural languages, which we suspect are more common for legacy systems than for contemporary ones. Because object-oriented languages are commonly researched, there is a large body of existing work we can build upon.

1.1.2 Models concurrency accurately

Because concurrency has become common and is architecturally significant, support for concurrency is useful in a system for reconstructing dynamic software architectures. Despite this, [4] found that distributed systems and multithreaded applications were among the least common targets for dynamic analysis.

1.1.3 Models the behavior of the system from software execution data

As explained before, analyzing the dynamic architecture has several advantages over analyzing the static architecture of a system. Most architecture models based on event logs are models of the dynamic architecture. However, a model of the dynamic architecture is not automatically a model of a system's behavior. For example, [7] creates a UML Component and Connector view, which does not model behavior but does represent the system at runtime.

Modeling behavior can be used for a variety of purposes, like debugging and understanding legacy code [8] or visualizing feature interactions [9]. Aside from creating understanding, models of software behavior can be used to analyze performance as well [5].

1.1.4 Doesn't require modification of the source code for instrumentation

Dynamic architecture reconstruction is based on execution traces. To get execution traces from a running system, the system must be *instrumented*: it must be changed so that it logs events like method calls. Manually adding more fine-grained logging to a large software system would take unacceptably long and thereby defeat the purpose of automatically reconstructing its

[1] https://www.tiobe.com/tiobe-index/, https://stackify.com/popular-programming-languages-2018/, http://pypl.github.io/PYPL.html.

architecture. Manually modifying the source code of a system to instrument it is therefore not desired.

1.1.5 Does not require knowledge of the source code

The purpose of automating SAR is to save developers time, especially those that are unfamiliar with the inner workings of the system under analysis. If an approach requires intimate knowledge of the source code of the system under study, it is not useful for people who are unfamiliar with the system. This reduces the applicability of the approach and makes the approach more difficult and costlier to use. Therefore, someone who has little to no knowledge of the code base should be able to create a visualization of the architecture and understand it.

1.2 PROBLEM STATEMENT

Our objective is to provide insight into the actual concurrent behavior of software within a software architecture derived from software execution data.

2 RESEARCH APPROACH

2.1 RESEARCH QUESTIONS

In the introduction, we have laid out five objectives for our software architecture reconstruction approach. These objectives combined lead to our main research question:

> MQ: What is a software architecture reconstruction approach that visualizes the dynamic architecture of an operational concurrent system with minimal developer effort?

To answer this main question, we pose a set of sub-questions that explain and decompose our main question. These questions are as follows:

> SQ1: What is the current state of the art in dynamic architecture reconstruction?

To create an understanding of the current state of the art, we look for approaches that have similar objectives to ours. We will use the findings from answering this sub-question to further identify the gap left by current approaches, as well as creating a basis for our own approach.

This is a literature question and will be answered through a literature review. The main method is snowballing from sources that are available online. Where possible, surveys and reviews of existing literature are used to maximize the breadth of the search. To make sure we do not miss the latest developments, we look for newer papers that cite influential papers as well.

The result is an overview of the current literature on SAR for concurrent, object-oriented programs that focus on the dynamic architecture.

> SQ2: What data needs to be collected to discover concurrent behavior from running software and how?

One of our objectives is to use software execution data from software to reconstruct its architecture. Before we can do that, we need to establish what data needs to be collected to create effective models. We also need to create a non-intrusive way of registering this data.

The result will be a framework for logging software execution data, as well as a system for registering this data. To create our framework and system, we use a design science method as described in Section 2.2. Once we have found a way of creating accurate event logs, we need to find a way to create a model from these logs. This leads to our third sub-question:

> SQ3: How can we visualize software execution data in software architecture?

We have established that we want to use logs of software to discover and visualize behavior. In SQ3 we present an approach for discovering the behavior of object-oriented systems from the data defined in SQ2. We will also present a visualization in which the object-oriented aspects and the behavior are simultaneously displayed.

> SQ4: Is the proposed approach feasible in real-life systems?

After creating our approach, we will consider its feasibility as a system for software architecture. The results will be the results of the assessment, an interpretation of these results and recommendations for future research.

2.2 RESEARCH METHOD

The basic method used for this research is a design science approach. The basic process is based on the design science research process (DSRP) of [10].

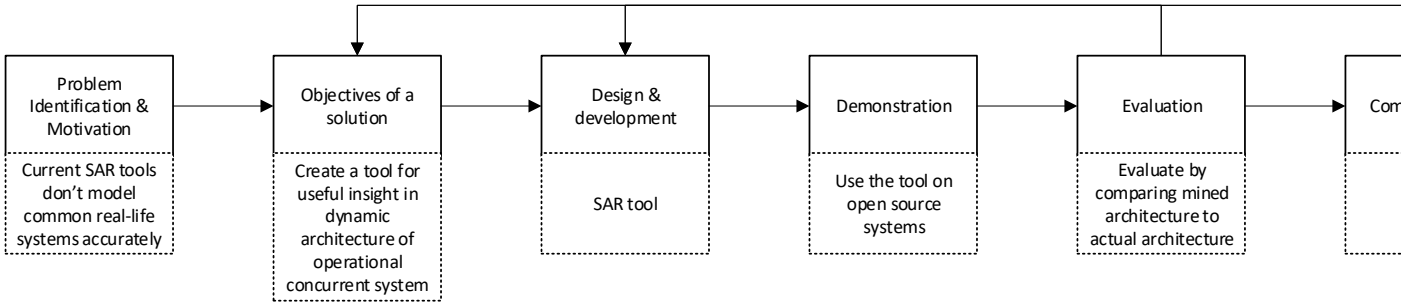


Figure 1: Method overview, adapted from [10]

2.2.1 Steps

The method shown in Figure 1 is divided into six steps. In the Introduction (Chapter 1) and Background (Chapter 3), we cover the problem identification & motivation and the objectives of the solution. These steps relate to SQ1. Chapter 4 and Chapter 5 cover the design & development step in the method, answering SQ2 and SQ3 respectively. The fourth step, Demonstration, is done through case studies. These case studies are covered in Chapter 6 through 8. The evaluation step is covered in the case study chapters and discussed in Chapter 9. The final step, Communication, is essentially covered by the entirety of this thesis.

3 BACKGROUND

In this chapter, we explain the context of our research. It relates to the first research question:

> What is the current state of the art in dynamic architecture reconstruction?

We will first cover software architecture in more detail. After that, we explain some of the formalisms we came across in our literature research and elaborate on how software behavior can be registered so that concurrent behavior can be derived. Finally, we present an overview of related approaches from literature and how they relate to our objectives and the concepts we explained.

3.1 SOFTWARE ARCHITECTURE

According to Bass, Clements and Kazman, "[the] software architecture of a program or computing 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" [11]. Similarly, Rozanski and Woods define the architecture of a (software) system as "… the set of fundamental concepts or properties of the system in its environment, embodied in its elements, relationships, and the principles of its design and evolution" [1].

By providing an abstraction of a system, software architecture reveals certain properties while hiding others [3]. According to Garlan, software architecture can play an important role in at least understanding, reuse, construction, evolution, analysis and management [3]. Different stakeholders have different uses for software architectures and require different perspectives on the system. There is no one view that covers every stakeholder need, meaning different uses require different views.

Throughout this thesis, the terms 'dynamic software architecture' and 'static software architecture' or variants are used. According to Rozanski and Woods, the static structures in a software architecture define a system's internal design-time elements and their arrangement [1]. Conversely, the dynamic structure defines the system's runtime elements and their interactions.

Software architecture reconstruction (SAR) approaches can use various inputs to reconstruct a system. These inputs can be similarly divided: static, design-time elements such as source code constructs are omnipresent in SAR, while insight in the runtime behavior of software can best be gained from dynamic inputs [2], [12]. We also use the term *software execution data* to refer to inputs of dynamic origin throughout this thesis.

Software execution data is generated by the system at run-time, which means that dynamic software architecture reconstruction approaches typically give different results to static approaches: dynamic approaches only have events that happened during run-time as input, while static approaches can include any part of the system that is available at design-time. It is important to note that while the information distilled from dynamic and static inputs often overlaps, they are not subsets of each other. A static approach will capture all information that can be observed from the source code of a system, but no information that is only available at run-time such as object identities or performance data. A dynamic approach on the other hand

will not pick up on anything that is not being executed while software execution data is being recorded. Combining a static approach with a dynamic approach will therefore lead to the most complete picture of the system under study.

3.2 VISUALIZATION TECHNIQUES FOR SOFTWARE BEHAVIOR

Visualization techniques are an effective way to convey information about the architecture at hand. There are various visualization techniques in common use, each suiting various views. We will give a short overview of some visualization techniques that are relevant to our use case: visualizing the concurrent behavior of software.

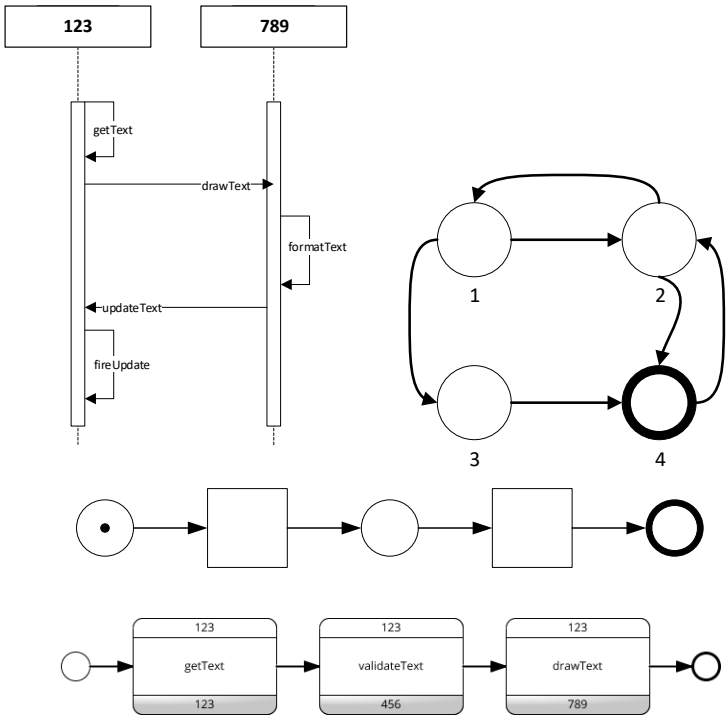


Figure 2: Four visualizations, from top left: MSD, FSM, Petri net, BPMN Choreography diagram

3.2.1 Message sequence diagrams

Message sequence diagrams (MSDs) are perhaps the easiest to comprehend of all modeling styles for software behavior. They present a timeline of different objects and the interactions between them. Concurrency is supported and can be indicated in an expressive manner. Crucially though, message sequence diagrams describe a *sequence*: whereas Petri nets and finite state machines (FSMs) can express different choices within a process, MSDs cannot. While MSDs are commonly used in architecture, they are not very useful in modeling different variations of the same process.

3.2.2 Finite state machines

Finite state machines (FSMs) are a formalism commonly used for representing behavior of automated systems. They are very low-level when not extended, consisting only of states the system can be in and transitions between those states. Every discrete state of a system is

represented in an FSM, making it an unsuitable modeling technique for systems with many states. Large, modern systems often have a nearly uncountable (if finite) number of states, so a finite state machine that covers all behavior must either be impractically large or very abstract.

FSMs in their basic form do not support any form of concurrency, constraints or hierarchy. As we will find in the following section, however, several extensions exist that add such features.

3.2.3 Models of concurrency

Like finite state machines, process models are a subset of the group of *transition systems*. Process models are used to represent processes in process mining and are commonly based on Petri nets. Whereas finite state machines do not support concurrency, Petri nets and its derived formalisms do. While simple, Petri nets also support constraints on when transitions can be 'fired', in contrast to finite state machines. As with finite state machines, extensions exist that add hierarchies.

While more powerful and therefore often more concise than finite state machines, process models that accurately represent the systems that they model can quickly become large and complicated. As we will find in the following section, there are ways of abstracting process models to make them more suitable for high-level analysis.

3.2.4 Choreography diagrams

Choreography diagrams were introduced in BPMN for representing choreographies. According to the W3C, "[a] choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state" [13]. Decker and Weske created a framework for choreographies, defining choreographies as being composed of interactions [14]. Every conversation is an instance of a choreography, and consists of messages between participants. Just like how a conversation is an instance of a choreography, a message exchange is an instance of an elementary interaction.

Choreography diagrams are similar to other BPMN process models and are less formal than for example Petri nets. On the other hand, they allow for more complicated behavior than message sequence diagrams, such as loops. Choreography diagrams are thus more high-level than message sequence diagrams, which only describe a single sequence instead of several possible sequences.

3.3 DETECTING CONCURRENCY

In a cursory investigation of existing SAR approaches, several approaches were found that had similar objectives to ours. However, most existing automated approaches had limitations in their approaches to concurrency or did not support concurrency at all. Process mining approaches do support concurrency, but their way of handling concurrency is limited. In process mining, concurrency exists in two forms:

- Process instances, which are marked separately in the event log
- The process that is being inferred

As we will show, both are problematic when creating accurate models of concurrent software. First, however, we provide a short background on event logs in process mining.

3.3.1 Event logs and process mining

In process mining, event logs are loosely defined to be logs of detailed events in business systems [15, p. 8]. While there are several process mining tools on the market, most of the research we found involved the open source framework ProM [16]. ProM uses the XES standard for its event logs[2]. In XES, event logs are log objects which contain an arbitrary number of traces. Each trace in turn contains any number of events. A trace is analogous to a case or process instance.

XES provides an open source standard (OpenXES) that defines a way in which behavior can be represented in XML (eXtensible Markup Language). In ProM, process objects adhere to the XES standard and can be used by ProM plugins to discover, analyze and visualize processes. Events in XES can have attributes that define properties of the event, like their name, who performed or initiated the event and the time at which the event happened.

3.3.2 Concurrency through process instances

In process mining, process instances or traces are sequences of events that together form a distinct instance of a process. In a process of providing an insurance for example, the process of providing a single insurance would be a trace. Despite being serialized as sequences, different cases can happen at the same time, and process mining tools are designed to handle these separate process instances separately.

The problem with using process instances as a mechanism for concurrency is that process instances are assumed to be isolated from each other. In computer systems, individual threads and processes cannot be accurately modeled as cases: different threads can interact and share resources. Conversely, in process mining, everything that happens in a process instance only affects that process instance. Therefore, the concurrency within a software system cannot be fully modeled, and a different approach needs to be found.

3.3.3 Concurrency within process models

In process mining, the concurrency of events within the process itself is inferred using heuristics. This is to overcome a limitation in process mining theory and the XES standard where events happen at singular points in time and are totally ordered. To explain the difference between totally ordered events and partially ordered ones, we take Figure 3 as our real model that we wish to reproduce from a log.

[2] http://www.xes-standard.org/

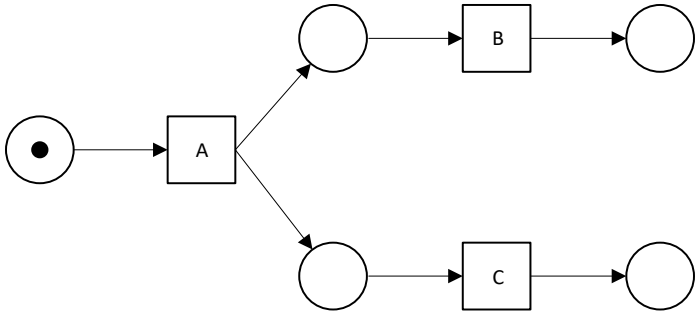


Figure 3: Petri net example of a model that cannot always be inferred using heuristics

The partial order of this model can be described as $A < B, A < C$: A happens before B, and A happens before C. Whether C happens before or after B is not known: they are assumed to happen at the same time. In process discovery algorithms such as the α-algorithm the assumption is made that if and only if two events follow each other in arbitrary order, they are concurrent [15, p. 130]. Whenever there is ambiguity within the ordering of events in the event log, this is interpreted as concurrency.

To satisfy an algorithm like the α-algorithm we need to present it two traces, such as $\langle A, B, C \rangle, \langle A, C, B \rangle$. However, there is no guarantee that within an event log both situations will occur and that the mining algorithm will pick up on this concurrency. It could be the case that transition B always happens before C because one server consistently responds quicker than another. It could also be the case that filtering is applied and either of the two options did not occur enough times to be considered 'significant' to the mining or filtering algorithm.

Summarizing, the concurrency in the partial order ($A < B, A < C$) is explicit: we know that in one process, B and C can both occur after A and their order relative to each other is not relevant. When using traditional techniques, concurrency is left implicit. It is assumed that because there exist similar processes where B follows A and C follows A, B and C must be concurrent. Because this assumption does not always hold, implicit concurrency reduces the accuracy of the model. Explicit concurrency is therefore preferable over implicit concurrency.

3.4 SIMILAR APPROACHES

To show that this thesis provides a contribution to the current state of the art, we review the available scientific literature in software architecture reconstruction (SAR). The main question we try to answer is the same as SQ1: What is the state of the art of modeling software from event logs?

The literature in this section was mostly found by searching Google Scholar for relevant literature and snowballing backwards and forwards from there on. By looking for newer papers that cite relevant literature, we try to find the most recent literature in every field.

Several reviews and surveys have been written on methods for software architecture reconstruction. One often-cited work is the review article written by Ducasse and Pollet [2]. This article gives a bird's eye view of software architecture reconstruction. It aims to provide engineers looking for an SAR solution with a taxonomy of methods, concepts and programs. It is somewhat dated (published in 2009) but lists an impressive number of solutions.

A similar article is Canfora et al.'s 2011 review article on software reverse engineering [6]. The term 'software reverse engineering' can be seen as interchangeable with software architecture reconstruction: Canfora et al. state that software reverse engineering aims to create 'high-level representations for an existing software system to support comprehension and evolution' [6]. This definition closely matches what is commonly meant by software architecture reconstruction.

Another overview is provided by Cornelissen et al. in the form of a survey of research on the *dynamic* analysis of software [4]. It provides an extensive overview of the different techniques discussed in the literature, which makes it easy to find systems that match our requirements.

In this chapter, we consider a small assortment of techniques found in the literature. Most of these techniques were published after the aforementioned reviews, with the exception of [7], [17], [18]. We divided the selected works into three categories: FSM-based approaches, process mining-based approaches and software architecture reconstruction approaches. The first two are approaches concerned with mining accurate models of behavior from event logs. The software architecture reconstruction subsection covers approaches that provide information on an architectural level.

3.4.1 FSM-based approaches

The two approaches here were chosen because they are recent, relevant and move the state of the art forward. FSMs were perhaps the first models reconstructed from traces [19] and they are still being actively researched.

A relatively recent approach is that of Walkinshaw et al. [20]. In this paper the authors provide a new way of creating finite state machines from execution traces. Another FSM approach, GK-tail (and therefore also GK-tail+, see [21]) has issues with determinism and flexibility. Walkinshaw et al. improve upon that approach by improving the process of finding guards. Guards are essentially conditionals that support data within a modeling language. They can only allow certain transitions to fire if a condition in the data is met, for example, if a variable contains a certain numerical value.

Because extending FSMs with data (for use with guards, for example) requires a data classification algorithm, Walkinshaw et al. need a classification algorithm as well. In GK-tail this was Daikon, which requires unrealistic amounts of data for accurate models [20, p. 821]. Additionally, it leads to non-deterministic models because states are inferred independently. Walkinshaw et al. make this classifier modular, so that they can use the many classifiers in WEKA.

The inference method as well as the data classification method use decision trees to create models. Like [21], the authors generate negative traces to test their approach. These negative traces are only invalid in their data, not in their events. They represent behavior that would not be recognized as invalid by a non-guarded system. This is a validation measure to check if the inferred model behaves in the same way as the original system. These negative traces also serve a second purpose: they ensure that a model will not overfit [15, p. 186].

A limitation of any system that uses guards to determine control flow is that the applicability is limited by the guard mechanism. This system and its process model counterpart [22] are both

limited to the primitive types of common programming languages. Modern languages like Java support an infinite amount of data types with a similarly infinite amount of conditions. They are not limited to string and equality and numerical comparison, which are the guard mechanisms supported by [20].

The second of our two FSM-based approaches is Beschastnikh et al.'s CSight, short for concurrent insight [8]. They create communicating finite state machines from traces so that concurrent systems can be analyzed. It is a low-level approach like that of Walkinshaw et al. [20], and it does not take objects into account.

CSight is unique compared to other approaches because it uses traces that are partial orders. Other approaches, as mentioned by [8] assume that traces are sequential. The partial ordering of events in CSight is defined using vector timestamps. Vector timestamps are the time indications used by a vector clock. This system was originally independently presented by Fidge and Friedemann [23], [24]. It is a logical clock system where every process keeps a vector of clocks for every other process in the system, with each process sending their latest vector with each message and updating their own vector when receiving a message. The system has no relation to conventional timestamps containing seconds, minutes and hours.

In the case studies provided by Beschastnikh et al. three different network systems are analyzed. Overall accuracy is good, but some edge cases remain.

3.4.2 Process mining-based approaches

The FSM approaches reviewed above provide very low-level representations of program behavior. To a degree, this defeats the point of software architecture. Moreover, the accuracy of some approaches is limited due to the lack of support for multi-threading. By contrast, process mining approaches generally revolve around target models with better support for concurrency, such as Petri nets and derived systems.

Process mining is a more recent phenomenon than synthesizing FSMs from traces. Introduced in the early 2000s, process mining aims to improve business processes [15, Ch. Preface] by creating process models from event logs.

While not the only paper on using process mining for reverse engineering software systems, that of M. Leemans and Van Der Aalst [5] is one of few that mentions it in the title. The technique described is novel in that it provides a way for reverse engineering distributed software into process models. It supports object-oriented languages, in this case Java. Still, it does not currently support multi-threaded software. The approach of Leemans and Van der Aalst can be placed 'in-between reverse engineering and process mining' [5] and thereby serves as a link between the two fields.

In addition to providing their own approach, Leemans and Van der Aalst compare 11 previous approaches to reverse engineering – most of which create UML Sequence Diagrams. Their own approach outputs a process model, without much static information to serve as context.

Together with Van Den Brand, Leemans and Van Der Aalst presented another approach for generating models of dynamic behavior of software [25]. Their paper provides a summary of 32 techniques used for this purpose, divided into four categories. The paper also presents a 33rd

technique, which aims to combine all criteria covered by the earlier papers. Like the previously mentioned approach of Leemans and Van Der Aalst ([5]), this technique aims to create process models from Java event logs using a process mining discovery algorithm. The process models in this newer approach are hierarchical process trees.

Hierarchical process trees are a new formalism, created specifically for hierarchical process trees. They are an extension of the process tree formalism that support sub-processes, creating a model that is a hierarchy of process models. Another innovation is – as the name suggests – that these new models can account for recursion. Statistically, the new mining algorithm scores about as high as the baseline, but also handles nested calls. In the conclusion, they present that their technique can integrate with Statechart and relate the diagrams to code in Eclipse. The latter option allows the hierarchical process tree to integrate with static architecture information.

Where Leemans and Van Der Aalst use hierarchical process trees, Liu et al. use hierarchical Petri nets [26]. In their paper, the authors present an approach for mining hierarchical Petri nets from software logs. The software logs are more detailed than most similar approaches. Like other approaches, the logs contain method calls, but in this approach, they also contain a calling method, an object and class ID for both the caller and the callee. The authors state that supporting explicit concurrency is one of the benefits of their approach, but they do not elaborate on how their approach achieves this.

The hierarchical Petri nets are not based on the static structure of the application, but instead on the call stack. A single process model (represented as a Petri net) is mined for each component. Components are defined manually, by creating a mapping of classes to components.

Another process mining approach for visualizing software behavior is that of De Leoni and Van der Aalst. In their approach, they are essentially applying a technique like that of Walkinshaw et al. ([20]) to Petri nets [22]. To do so, they introduce a new variant of Petri net: Petri nets with data. Because the control flow works in the same way as with Walkinshaw et al., the limitations on the supported control flow are similar. *In theory* all control flow is possible, however the logical operations on a given data type must all be able to be modeled in the decision tree that is being used. For future research, they propose using Daikon instead of their current classification method. However, Daikon is the algorithm Walkinshaw et al. try to steer away from to get more accurate models. This approach is limited in its application: it models software on a low level, like Walkinshaw et al. It does not use information about objects and does not explicitly support concurrency.

A problem with mined process models is that they can become very large when mined from a large and varied data set. As a solution, Van Der Aalst et al. presents a new method for mining processes [27]. They present a process discovery (mining) technique that uses localized events. This means the events in the event log have information about their *region*, and this information is used to improve the quality of the mining process. A region can be a service, a system, a component, etcetera.

Every event belongs to one or more regions. The algorithm that is presented mines a model (system net, a kind of labeled Petri net) for each region. Regions can only interact through events

that belong to both respective regions. After mining the individual region-based models, these models are combined into a global model by merging on these overlapping events.

In their paper, the authors formally prove that this approach leads to smaller models by around a factor of 10 [27]. They also do a case study on synthetic and real-life data and find that performance is often better using their new approach.

While the approach presented in the paper does not support asynchronous behavior, it claims that most of the formal proofs presented do hold in that case.

3.4.3 Software architecture reconstruction approaches

Software architecture reconstruction approaches focus on creating models relevant for software architecture. Often these models are UML diagrams, but can also include Functional Architecture Models or non-standardized or ad-hoc formalisms. Where for Petri nets and FSMs the formal aspects of the model are often considered, this is usually not the case for architecture models. Formal aspects include being able to prove reachability and correctness. Instead, the models referred to in this subsection are semi-formal or informal.

In this section, we review four software architecture reconstruction approaches. In the first approach, Van Der Werf and Kaats present an approach for using event logs to mine scenarios in Functional Architecture Models [9]. The inputs are traces consisting of both feature labels as well as module labels. These labels are used to reconstruct a FAM, which serves as the static structure of the model.

The proposed technique mines the dynamic aspects – communication between features and modules, as well as within modules. These are called the scenarios within a FAM. Moreover, it mines the static structure of the FAM using the module and feature information provided in the traces.

What sets the approach in this paper apart from other process mining approaches is that the behavior of individual modules is modeled, and composition techniques are then used to create an ‘overall’ process. This overall process is the scenario that is executed ‘over’ the FAM. As presented in the paper, the approach is still theoretical and has not been applied to real-life event logs.

The second approach we consider is DiscoTect, an SAR system that can mine the dynamic structure as well as design patterns from Java programs [7], [28]. These patterns must be defined first however, increasing the amount of developer involvement required. DiscoTect is designed around object-oriented languages and sees objects as unique rather than generalizing them to their class.

Note that DiscoTect creates a model of the dynamic structure, not of the actual processes. The resulting model is a component and connector view of the run-time structure of the program. This is different from most other tools that use event logs as an input which represent the behavior of the system. As a result, it is not possible to see what the order of execution is between components.

In the third approach, Walker et al. use traces to visualize the relationships between architectural components [17]. They manually create a mapping from software components to architectural elements. This mapping is then applied to a trace, allowing users to replay a trace on a block diagram showing the different architectural elements.

The logging process is more extensive than with most other approaches: they register class and instance entry and exit, object allocation and deallocation and thread start and stop. They abstract away object identifiers for their mapping process. To record class, instance and object information, the source code of the target system must be altered. This is a more involved process than the logging process for other methods.

To make trace information more insightful, they provide two aspects of abstraction: that of paths (abstracting through time) and that of abstract events (event types categorized to larger categories through manual mapping). Both can be related to the technique of Van der Werf and Kaats [9]: paths can be compared to scenarios and abstract events can be compared to modules and features.

Fourth and finally, Salah and Mancoridis present a hierarchy of dynamic software views [18]. These software views are dynamic in the same sense as the approach of [7]: the information pertains to the run-time aspects of a program. Yet, like with DiscoTect, they do not show in what order events occur in the system.

The dynamic software views introduced by the authors are the Object-Interaction View, Feature-interaction View, Class-interaction View and Feature-implementation View. The hierarchical aspect is how the information required for each view depends on that of another view. For example, object interaction is the basis for feature interaction and class interaction, while feature implementation in turn relies on the feature interaction and class interaction views.

To identify features, the system of Salah and Mancoridis requires users to use marked traces. Marked traces are created manually, by marking a moment and then using a certain feature. The events that are then logged ‘belong’ to that marked feature.

3.5 Conclusion

In this chapter, we sought to answer the first sub-question:

> What is the current state of the art in dynamic architecture reconstruction?

The state-of-the-art modeling software from event logs has many facets. One facet of the literature is creating formal models of software behavior. These approaches usually mine FSMs or process models of some sort. Approaches like those of [20], [22] focus on modeling the behavior on the level of control flow by including the conditionals that trigger certain flows.

Modeling the complete control flow is limited in terms of accuracy by the chosen formalism. Any formalism that can achieve 100% accuracy must support all control flow mechanisms of the source language. While this is possible, the state of the art has not reached this point yet. It can be argued that this situation is undesirable, because such a complex model would not be much more understandable than the source code.

More high-level formal models also exist. [8] describes an approach for creating concurrent FSM that improve the understandability of a network protocol. The authors' approach is also novel because they do not mine concurrency from the same source material as other approaches. By adjusting their instrumentation and mining vector timestamps, they can guarantee correct partial orderings instead of using heuristics. The latter is commonly used in process mining; its shortcomings are explained in the research problem analysis chapter of this thesis.

An approach similar to [8] is that of [5]. This approach also mines models from distributed systems, although these systems are represented as process models – essentially Petri nets with labeled transitions and time information. Like all process mining algorithms but unlike [8], [5] use totally ordered data as input and models concurrency using heuristics.

With any model, the level of abstraction affects its usefulness. A lot of process mining literature concerns the simplification of mined models to make them more understandable. This includes [29], where a mining algorithm is presented that accounts for the fuzzy nature of real-life processes. To do this, similar processes are simplified as being identical. While the mostly deterministic nature of computer programs means that such an approach is not very useful, there are other approaches for simplifying models. [30]–[32] create hierarchies from mined models, similar to how software architectures are often hierarchical. [25] is such an approach as well, but in that method, hierarchies are also used to model recursion.

While superficially similar to the above hierarchical approaches, [27] simplifies complex process models in a different approach. By using region information present in the event log, more accurate and simpler models can be created. While most other approaches simplify models after they have been mined, this approach leverages the available information to immediately get better results.

The other main facet represented in this chapter is creating informal and semi-formal models of behavior. In our case, we look into how these models a software architecture reconstruction context. [9] for example creates a functional architecture model, simultaneously integrating the static architecture of the system by mining components and features from event logs. The dynamic aspect visualized is the flow of control between the features in the system. A practical application of a method similar to that of [9] is that of [17], where extensive logs of a Java application are used to create visualizations. These visualizations show which code units interact in the system.

The aforementioned systems all create models of behavior, but other dynamic views can also be considered. [28] for example present DiscoTect. DiscoTect uses event logs to recognize pre-defined architecture patterns within a system. Models created with DiscoTect not only include these patterns, but also the different objects and classes involved as unique entities. This is useful information that cannot be found using static SAR methods.

Similarly, [18] present the use of event logs to create dynamic models. They present views like class interaction, feature interaction and feature implementation. Identifying features however requires manual work: to do so, they use marked traces. Not all the views in this approach require features to be manually identified, however. Finally, the approach presented in [26] falls into two categories. Hierarchies are used to abstract the process model into call stacks. Besides this

application of hierarchies, this approach also allows users to map architectural elements to parts of the system under study. This allows users of the approach the bridge the gap between software classes and architecture.

3.5.1 Overview

By considering a wide variety of different methods presented in literature, we find that no system exists that takes advantage of runtime information in all the ways we would like to. To recap, we had the following objectives:

1. Represents the object-oriented aspects of software
2. Models concurrency accurately
3. Models the behavior of the system from software execution data
4. Doesn't require modification of the source code for instrumentation
5. Doesn't require knowledge of the source code

In Table 1, and overview of the SAR approaches discussed in this chapter is provided. Because all approaches meet objectives 3 and 4, we left those out of the table. Based on Table 1, we can see that none of the systems we analyzed meets all our objectives outright. We aim to combine these techniques and meet all our objectives, thereby advancing the state of the art in terms of SAR.

	Authors	Basic Formalism	Formalism Variation	1	2	5
[5]	Leemans and Van Der Aalst	Petri net	w/ Performance data	Partial	Partial (t.o.)	Yes
[25]	Leemans et al.	Process tree	Hierarchical …	No	Yes (t.o.)	Yes
[20]	Walkinshaw et al.	FSM	Extended	No	No	Yes
[22]	de Leoni and Van Der Aalst	Petri net	w/ Data	No	No	Yes
[9]	Van Der Werf and Kaats	FAM	w/ Scenarios	No	Yes (t.o.)	Yes
[7]	Schmerl et al.	C&C view		Yes	No	No
[8]	Beschastnikh et al.	FSM	Communicating	No	Yes (p.o.)	Yes
[17]	Walker et al.	'Visualization'	Architecture level	Yes	Partial (threads)	No
[27]	Van der Aalst et al.	Petri net	System net	Partial	Partial (t.o.)	Yes
[18]	Salah and Mancoridis	Self defined	Architecture level	Yes	No	No
[26]	Liu et al.	Petri net	Hierarchical …	No	Yes (p.o.)	No

Table 1: Overview of existing dynamic SAR approaches.

[26] do not truly support object-oriented aspects of a system, but regions can be mapped to classes or components. [5] do use the class names and structure in labels, but do not incorporate the object-oriented structure of the program in their models. [26] require pre-defined features to be associated with events. (t.o.) = based on totally ordered event logs. (p.o.) = based on partially ordered event logs. (threads) = identifies individual threads in event logs.

4 COLLECTING CONCURRENT SOFTWARE BEHAVIOR

In this chapter we elaborate on software execution data, how it can be captured and how this is implemented in our approach. Our architecture reconstruction approach is based on software execution data. Software execution data is any data pertaining to the execution of a software system. This can be an event log, but also information about exceptions, requests, memory usage: any run-time information that can be registered.

The most common way of recording software behavior is recording method or function calls. Depending on the programming language being used, either the word *function* or *method* is used. In Java and C# all callable units of instructions are called methods instead of functions, so for the purpose of this thesis we will stick to 'method'. The terms *function* and *method* are in practice interchangeable.

Some approaches for recording software behavior record only the original call or *method entry*, while other approaches record both method entry and exit. Method entry here is the moment a method is called. The method can then call other methods. Method exit is recorded either when the method returns or when it stops without returning anything.

4.1 INTERACTION LOGS

We record the execution data of software as a bag of finite sequences of interactions. Formally:

$$L \subseteq \mathbb{B}(\langle \mathcal{O} \times \mathcal{M} \times \mathcal{O} \rangle^*)$$

Where:

- $\mathbb{B}(X)$ denotes the set of all possible bags over set X
- $\mathcal{O}$ is the set of all objects in the log
- $\mathcal{M}$ is the set of all messages in the log
- Given an interaction (a, m, b), we define $\pi_{caller}\big((a, m, b)\big) = a, \pi_{callee}\big((a, m, b)\big) = b, \pi_{message}\big((a, m, b)\big) = m$.
- We assume that for any $a, b \in \mathcal{O}, m \in \mathcal{M}, (a, m, b)$, m is a method of b. Static methods belong to a single 'static' instantiation of their respective classes.

The final point is linked to the static architecture of the system from which the log is derived. Every interaction in the log is a method call. The message is then the name of the method in the callee class that is being called. Unlike [26] we don't register the calling method; we assume that every method call is registered and that the calling method is the method that precedes this call.

As a running example for the theoretical part of this thesis, we consider a very short, hypothetical log. The following table shows a representation of this log, which we will be using to create an example of a hierarchical interaction model. Our example interaction log holds no concurrent behavior and consists of just seven interactions. We assume Java-like fully qualified names for the objects. Each object is identified by a unique ID.

ID	Caller	Message	Callee
1	org.Model.123	getText	org.Model.123
2	org.Model.123	validateText	org.Validator.456
3	org.Model.123	drawText	com.ui.GUI.789
4	com.ui.GUI.789	formatText	com.ui.GUI.789
5	com.ui.GUI.789	updateText	org.Model.123
6	org.Model.123	fireUpdate	org.Model.123
7	org.Model.123	fireUpdate	org.Model.321

Table 2: Running example log

Table 2 shows the sequence S where $s_1 \dots s_n \in S, n = \|S\| = 7, L = [S]$. The first element in the log, $s_1 = (\langle org, Model, 123\rangle,\ getText, \langle org, Model, 123\rangle)$. The second element, $s_2 = (\langle org, Model, 123\rangle, validateText, \langle org, Validator, 456\rangle)$, et cetera.

4.2 IMPLEMENTATION: AJPOLOG

As a proof of concept, we implemented our proposal in a tool called AJPOLog[3]. The name AJPOLog comes from AspectJ Partially Ordered Logging as it uses AspectJ to instrument systems. Put briefly, AspectJ is a system for 'weaving' code into existing Java code that is often used for instrumenting Java software [5, p. 3], [7]. An advantage of this approach is that the original source code does not have to be adjusted to register software execution data; it is non-intrusive at the source level.

Partially Ordered refers to the mathematical concept of partial ordering that we marked as a necessity in our proposed approach. Every event registered by AJPOLog is a call from an object to another, consisting of the calling object ('caller'), the method being called ('message') and the object to which the method belongs are logged. Objects are identified by their fully qualified name (FQN) as well as their `identityHashCode`. The `identityHashCode` is used as an ID for each object, as it should stay the same across the lifetime of an object[4]. In case of a static method call, the ID is replaced by the string 'Static'.

By explicitly logging the flow of control and uniquely identifying the objects involved, we attain a partial order of method calls. A new event is created every time a method is either entered or exited. The full list of log entry information is as follows:

1. Timestamp
2. Name of the thread on which the call happened
3. Whether the event refers to a method entry or exit
4. Identifier (FQN + object identityHashCode) of the calling object
5. Identifier of the object that contains the method being called
6. Method signature and fully qualified name of the method being called

[3] AJPOLog and related tools can be found at https://github.com/tijmendj/AJPOLog
[4] The identityHashCode and its overridable counterpart HashCode are not unique across all objects: see Section 9.2.1.2

Apache Log4j2[5] is used to create a log of all events created by the aspect code. By using a dedicated logging framework, we save ourselves the trouble of handling high-performance, reliable writing of log information ourselves.

In the design of the logging aspect, a few considerations have been made to reduce the performance impact of instrumentation. Namely, all calls are sent to Log4j immediately: the aspect itself does not keep any of them in memory. String modification operations are also kept to a minimum.

```
2018-11-27T15:28:36,588;[main];Entry;org.architecturemining.program.example.band
.BandPractice.Static;java.util.ArrayList.CallerPsuedoId: 1337344609;public
boolean java.util.ArrayList.add(java.lang.Object)
2018-11-27T15:28:36,593;[main];Exit;org.architecturemining.program.example.band.
BandPractice.Static;java.util.ArrayList.CallerPsuedoId: 1337344609;public
boolean java.util.ArrayList.add(java.lang.Object)
2018-11-27T15:28:36,593;[main];Entry;org.architecturemining.program.example.band
.BandPractice.Static;java.util.ArrayList.CallerPsuedoId: 1337344609;public
boolean java.util.ArrayList.add(java.lang.Object)
2018-11-27T15:28:36,594;[main];Exit;org.architecturemining.program.example.band.
BandPractice.Static;java.util.ArrayList.CallerPsuedoId: 1337344609;public
boolean java.util.ArrayList.add(java.lang.Object)
```

Figure 4: The first four lines of an example log (lines wrap to fit the line width)

4.2.1 Conversion script

Part of the process of extracting logs from applications is running the logs created by the aspect through a script. This script is included in the AJPOLog project and mainly serves to pre-process the 'raw' data into a CSV file that ProM can easily read. An excerpt from such raw data is depicted in Figure 4. It works as follows:

1. When running the script, the user is required to give the filename of the log that is to be converted. This script opens reads this file line-by-line.
2. Because every line in the log either refers to a method entry or exit, every entry is pushed to a stack.
3. When an exit is encountered, the script pops the corresponding entry from the stack
4. The information regarding to the method is processed to fix any missing callers (explained in the following subsection).
5. Once the method call is processed, the entry time, exit time and the remaining information are written to a CSV file in the correct format.
6. Once all lines in the input file are processed, the stack is flushed. If there are remaining entries they are popped from the stack and given an exit time of 2999-12-31T00:00:00,000.

For simplicity, one aspect of the script is left out of the above description. The script is built to handle multithreaded code and therefore keeps one call stack for each thread instead of one global call stack.

Something we found is that for a log to be properly processed by ProM's CSV-to-XES plugin, the CSV file needs to always have a header row with column labels. Furthermore, we found that ProM

[5] https://logging.apache.org/log4j/2.x/

could auto-detect most options when using semicolons as separator characters and the ISO 8601 format for date and time. AJPOLog therefore formats the resulting CSV files as such. An example is presented in Table 2. A full width version is available in Appendix A.

Start Time	End Time	Thread	Caller	Callee	Message
2018-11-27T1	2018-11-27T	[main]	org.archite	java.util.A	public boolean java.util.ArrayList.add(java
2018-11-27T1	2018-11-27T	[main]	org.archite	java.util.A	public boolean java.util.ArrayList.add(java
2018-11-27T1	2018-11-27T	[main]	org.archite	java.util.A	public boolean java.util.ArrayList.add(java
2018-11-27T1	2018-11-27T	[main]	org.archite	java.util.H	public java.lang.Object java.util.HashMap.

Table 3: First four lines of the same log, converted to CSV

4.2.2 Using AJPOLog

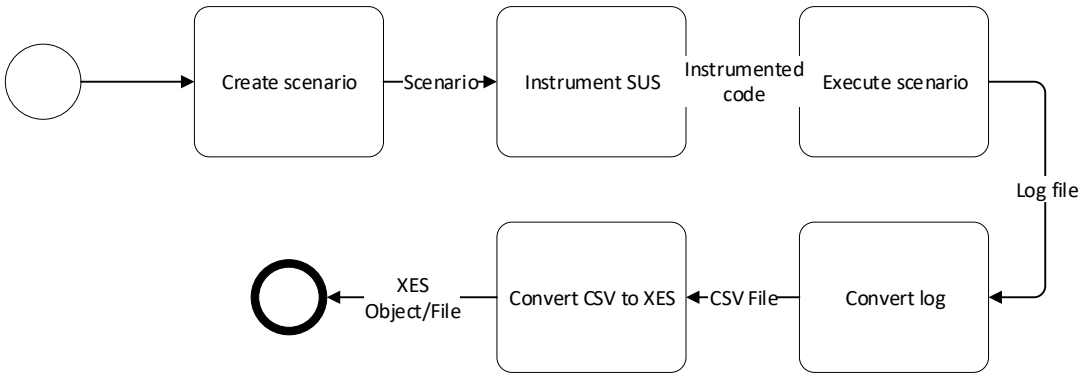


Figure 5: Overview of the AJPOLog instrumentation process

Using AJPOLog is a five-step process. First, a scenario must be created. The scenario defines the set of use cases of the system under study (SUS) that will be analyzed. Creating a suitable scenario is important, because only methods that are called during the execution of a program will appear in the log. If the SUS is for example a word processing program and the scenario doesn't contain the print functionality, there is a good chance none of the printing methods will be found in the log.

Second, the system-under-study must be set up to weave our instrumenting aspect. We use the Eclipse IDE for weaving aspects as we found it more convenient than doing so using command line tools. The third step is to execute the scenario on the instrumented system, thereby generating a log file.

The fourth step is to use our conversion script, converting the raw log to a CSV file that can be read be ProM. Fifth and finally, this CSV file is converted into an XES object in ProM, allowing our data to be used by a plethora of process mining plugins, including our own.

4.3 Conclusion

In this chapter, we answer sub-question 3:

> What data needs to be collected to discover concurrent behavior from running software and how?

The answer to this question is that we register the method calls of a system as partial orders of the form $caller < callee$. The callers and callees are represented by their fully qualified names,

including their containing packages and namespaces, their class name and instance ID. This way the concurrency of method calls is preserved, and we do not have to use heuristics when mining concurrent behavior.

After establishing a set of requirements for a method for capturing concurrent behavior from running software, we implemented our own tool for doing so: AJPOLog. AJPOLog stands for AspectJ Partially Ordered Logging, owing to its use of AspectJ for instrumenting Java programs. While AspectJ has some drawbacks, the tool is still capable of capturing reasonably accurate software execution logs. By using a conversion script, we convert the raw log file into a CSV file that can be read by ProM and used for process mining. This method is non-intrusive: the original source code is not modified to create software execution data.

5 MODELING SOFTWARE BEHAVIOR

In this chapter, we will answer sub-question 3:

> How can we visualize software execution data in software architecture?

Three of our objectives relate to SQ3, namely:

- Representing the object-oriented aspects of software
- Modeling concurrency accurately
- Modeling the behavior of the system from software execution data

Starting with the first requirement, UML class diagrams seem like an obvious choice. They show the exact class structure of an object-oriented program and support the full range of relationships and elements supported by languages like C++ and Java. However, the second and third requirements are both at odds with a static view like a class diagram. A class diagram does not show the behavior of software, neither in sequential nor concurrent behavior.

We covered several different visualization techniques in Chapter 3. One approach that relates to ours is that of Van Der Werf and Kaats [9], wherein the authors extend functional architecture models to include scenarios. These scenarios describe partially ordered interactions between the elements in a functional architecture model. Using these scenarios, it is possible to indicate behavior in what was previously a static view. Furthermore, because the interactions are partially ordered, concurrency can be properly displayed.

Functional architecture models are made up of features, which are contained in modules. A module can contain several features and can be contained by another module. The structure of modules and features is a tree, where leaf nodes are features and all non-leaf nodes are modules. Features can be connected with directed edges called information flows or interactions. These information flows are labelled with the type of information flow that happens.

Where functional architecture models represent a system from its usage perspective, we want to represent systems from an object-oriented, behavior-based perspective. In a FAM, edges represent possible interactions between features. We see these interactions as analogous to the interactions – method calls – between objects in our logs. Scenarios as presented in [9] are then ordered, realized interactions: processes within the system under study. We propose a variation of functional architecture models: hierarchical interaction models, an object-oriented counterpart to functional architecture models.

Functional Architecture Model	Hierarchical Interaction Model
Usage perspective	Object-oriented perspective
Feature	Object
Module	Container
Information flow	Interaction edge
Feature or module	Component
Information flows go from features to and from features	Interaction edges go from and to components

Table 4: Main differences between functional architecture models and our models

In Table 4 we give an overview of how our models, titled hierarchical interaction models, differ from functional architecture models. Despite the structure being nearly identical, the basic idea of what is represented is quite different. Functional architecture models present the usage perspective, with the system divided up into functional elements. Hierarchical interaction models, conversely, are more low-level and represent an object-oriented, code-based perspective.

The reason behind this difference stems in the source material (software execution logs) and goal of our approach. Software execution logs normally do not have any functional labeling and adding that automatically would be beyond the scope of this project. Therefore, we stick to using the structure found in the source code. We presume that any reasonably well-written system has at least some structure in its design that aids in understanding it.

Having explained the main difference between the approaches, we can look at the structural differences. In functional architecture models, features are the leaf nodes of the feature and module tree: they are the most low-level elements. In hierarchical interaction models, objects are analogous to features, containers are like modules and both types of elements together are called components. The tree of features and modules in functional architecture models is directly comparable to the tree of components in hierarchical interaction models.

In functional architecture models features have information flows between them. These are directed edges just like interaction edges in hierarchical interaction models. Nonetheless, in hierarchical interaction models (HIM), these edges go from component to component. We defined components as a collective name for both objects as well as containers, and therefore, it is also possible to have interactions between containers and objects, or containers with other containers.

5.1 FORMAL DEFINITION

Formally, a hierarchical interaction model is a 5-tuple $(\mathcal{O}, \mathcal{T}, c, \mathcal{M}, \rightarrow)$ where:

- $\mathcal{O}$ is the set of objects
- $\mathcal{T}$ is the set of containers.
- $(\mathcal{O} \cup \mathcal{T})$ is the set of components.
- c is a function from $\mathcal{O} \rightarrow \mathcal{T}$, mapping objects to containers. Its transitive closure is irreflexive.
- $\mathcal{M}$ is the set of messages
- $\rightarrow \subseteq \mathcal{O} \times \mathcal{M} \times \mathcal{O}$, a subset of the set of all objects connected with all objects by all messages.

A container holds one or more objects and zero or more containers. The function c represents this mapping. Elements in $\rightarrow$ are the interactions that can occur in a system and feature a caller, a message and a callee. Just like with interaction logs, for $a, b \in \mathcal{O}, m \in \mathcal{M}$ an interaction is of the form (a, m, b).

A log L *conforms to* a HIM $(\mathcal{O}, \mathcal{T}, c, \mathcal{M}, \rightarrow)$ if it only contains interactions allowed by the HIM, i.e., if $L \subseteq \mathbb{B}(\rightarrow^*)$. Recall the example log in Table 2. We can infer the objects that are in $\mathcal{O}$ from the log by taking all values of $\theta \in \mathcal{O}$ that are in the log and listing them:

- `org.Model.123`
- `org.Validator.456`
- `com.ui.GUI.789`
- `org.Model.321`

We use the fully qualified name to denote an object. For example, org.Model.123 represents the sequence $\langle org, Model, 123 \rangle$. The fully qualified names of the objects in the event log induce a forest on the objects: each fully qualified name forms a branch in this forest. In a HIM, this forest is a visualization of $(\mathcal{O}, \mathcal{T}, c)$.

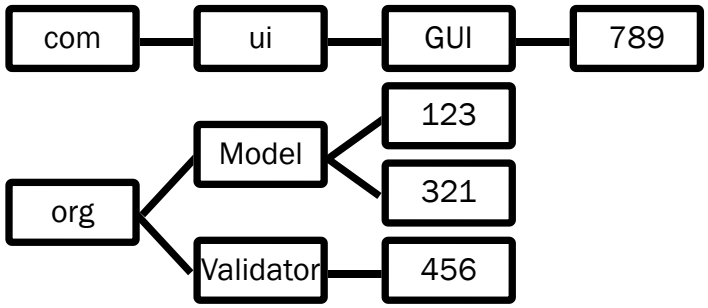


Figure 6: Running example component tree

In this forest, the component structure found in the log is represented. Any branch read from left to right in Figure 6 is a sequence of components that occurs within the log in Table 2. The leaf nodes are objects, and the remaining nodes are containers. In a similar manner, we can draw this tree in a HIM-style diagram. In Figure 7 we visualize $(\mathcal{O}, \mathcal{T}, c)$, by drawing all containers, $\mathcal{T}$, with dotted lines and all objects, $\mathcal{O}$, with solid lines.

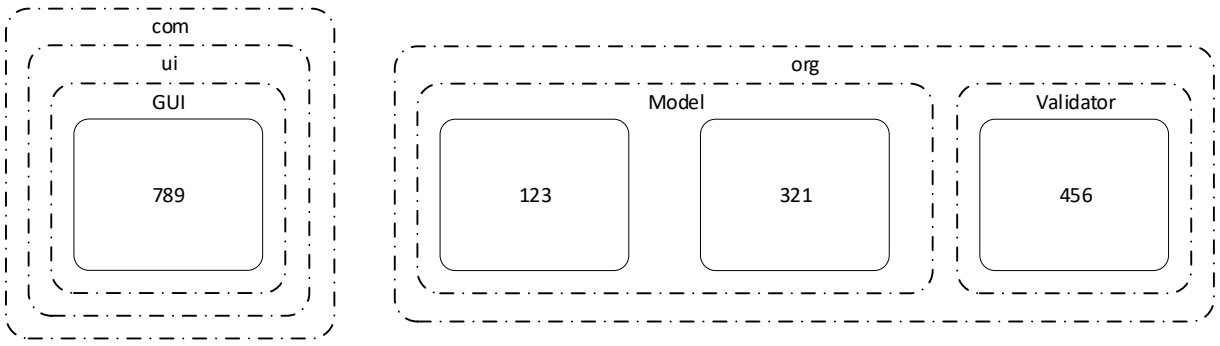


Figure 7: Running example component tree as a HIM

5.2 INTEGRATING PROCESS MODELS

While Figure 7 displays all the component information of the example log, it does not show any interaction edges. Every element in the log contains a caller, callee and a message, which we can re-interpret as edges. Figure 8 is a visualization of the entire graph, $(\mathcal{O}, \mathcal{T}, c, \mathcal{M}, \rightarrow)$.

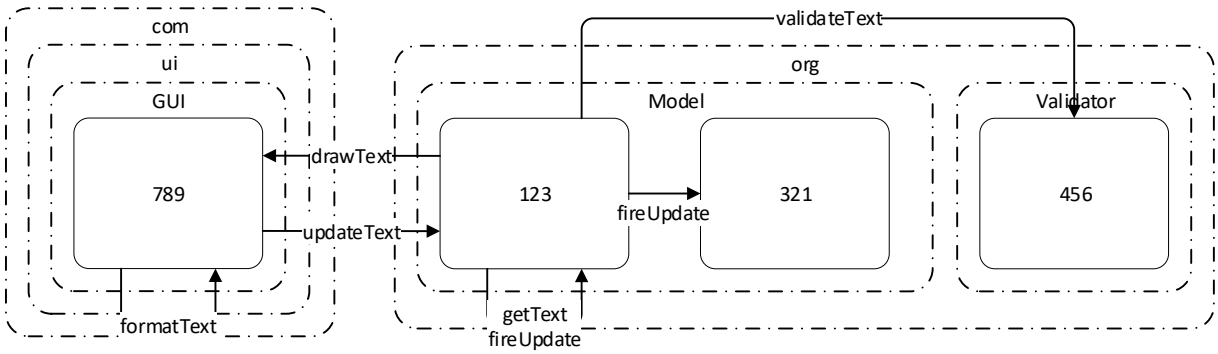


Figure 8: Running example component tree as HIM with interaction edges

Like FAMs in [9], we can visualize a scenario on our running example HIM. We also include a message sequence diagram in Figure 10. For brevity, we only show the last element of each hierarchy. The numbering corresponds to the sequence order of the entries, and the ID column in Table 2.

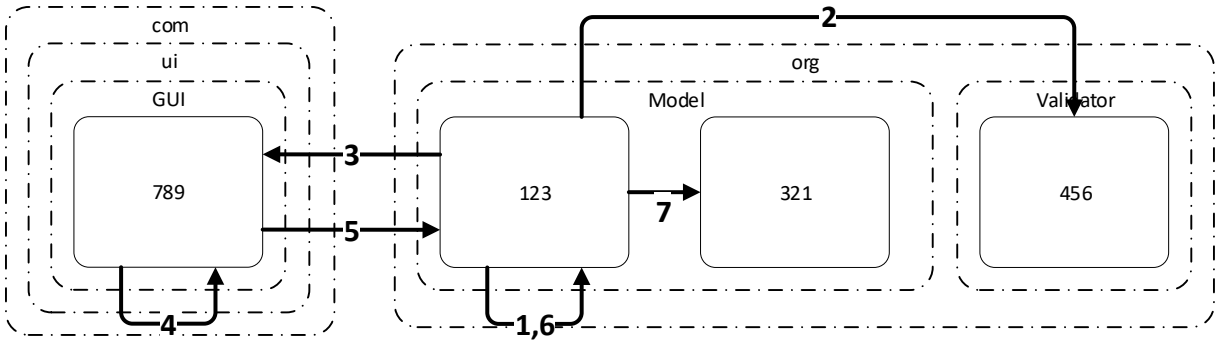


Figure 9: Running example HIM with scenario

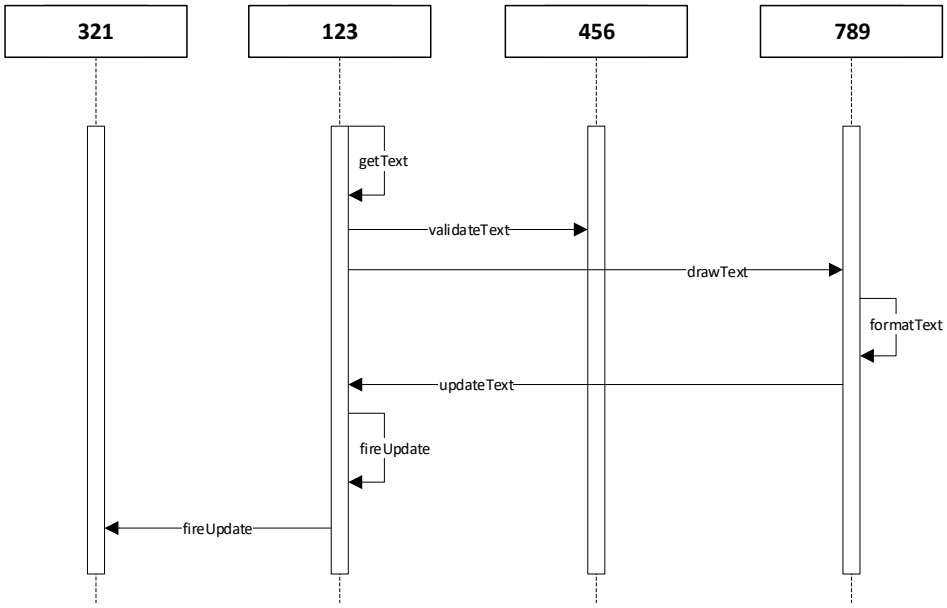


Figure 10: Running example message sequence diagram

5.3 SELECTING COMPONENTS

So far, we've considered a running example log with only seven entries. Real life logs are however much longer: the JabRef log we analyze later in this thesis, for example, has over 300,000 entries. A process model representing all that behavior would be far too large to analyze, which is why we want to be able to create models from a subset of all components in a log. To explain how this works, we will define what constitutes the behavior of a set of components.

We can recursively define the behavior of component c as all behavior of the component itself and that of its children. Let α and β be sequences, such as the component structures of a caller or callee. $\alpha \sqsubseteq \beta$ is then true if α is a prefix of β. Formally:

$$\alpha \sqsubseteq \beta \Leftrightarrow \forall 1 \leq i \leq \|\alpha\| \leq \|\beta\| : \alpha(i) = \beta(i)$$

If we want to know the behavior of all objects in a container, we can use this definition to create a set of objects that are contained, directly or indirectly, in a container. For example, all objects contained in *org.Model* are $(o \in O | \langle org, Model \rangle \sqsubseteq o)$.

If we want to explore the behavior of several components in a process model, we could investigate defining the behavior of a *set* of components. While it is possible to provide more than one sensible definition of the behavior of (a set of) components, we define the behavior of more than one component as all interactions where the selected components are prefixes of both the caller and the callee. Given a sequence σ we inductively define $\mathcal{B}(\sigma, c)$ as:

$$\mathcal{B}(\epsilon, C) = \epsilon$$

$$\mathcal{B}(\langle a, m, b \rangle; \sigma, C) = \begin{cases} (\langle a, m, b \rangle); \mathcal{B}(\sigma, C) & \text{if } a, b \in C \\ \mathcal{B}(\sigma, C) & \text{otherwise} \end{cases}$$

Where ϵ denotes the empty sequence, $\nu; \mu$ the concatenation of two sequences ν and μ, and C a set of components.

Consider the example log $L = [S]$ of Section 4.1. Then, the behavior of component $\langle org, Model \rangle$ can be expressed as: $\mathcal{B}(S, \{\langle org, Model \rangle\}) = \langle s_1, s_6, s_7 \rangle$ and for the interaction between components $\langle org, Model, 123 \rangle$ and $\langle com \rangle$ as: $\mathcal{B}(S, \{\langle org, Model, 123 \rangle, \langle com \rangle\}) = \langle s_1, s_3, s_4, s_5, s_6 \rangle$. The log of the latter example is shown in

Table 5, and a process model of their interaction is depicted in Figure 11.

ID	Caller	Message	Callee
1	org.Model.123	getText	org.Model.123
3	org.Model.123	drawText	com.ui.GUI.789
4	com.ui.GUI.789	formatText	com.ui.GUI.789
5	com.ui.GUI.789	updateText	org.Model.123
6	org.Model.123	fireUpdate	org.Model.123

Table 5: Behavior of com and org.Model.123

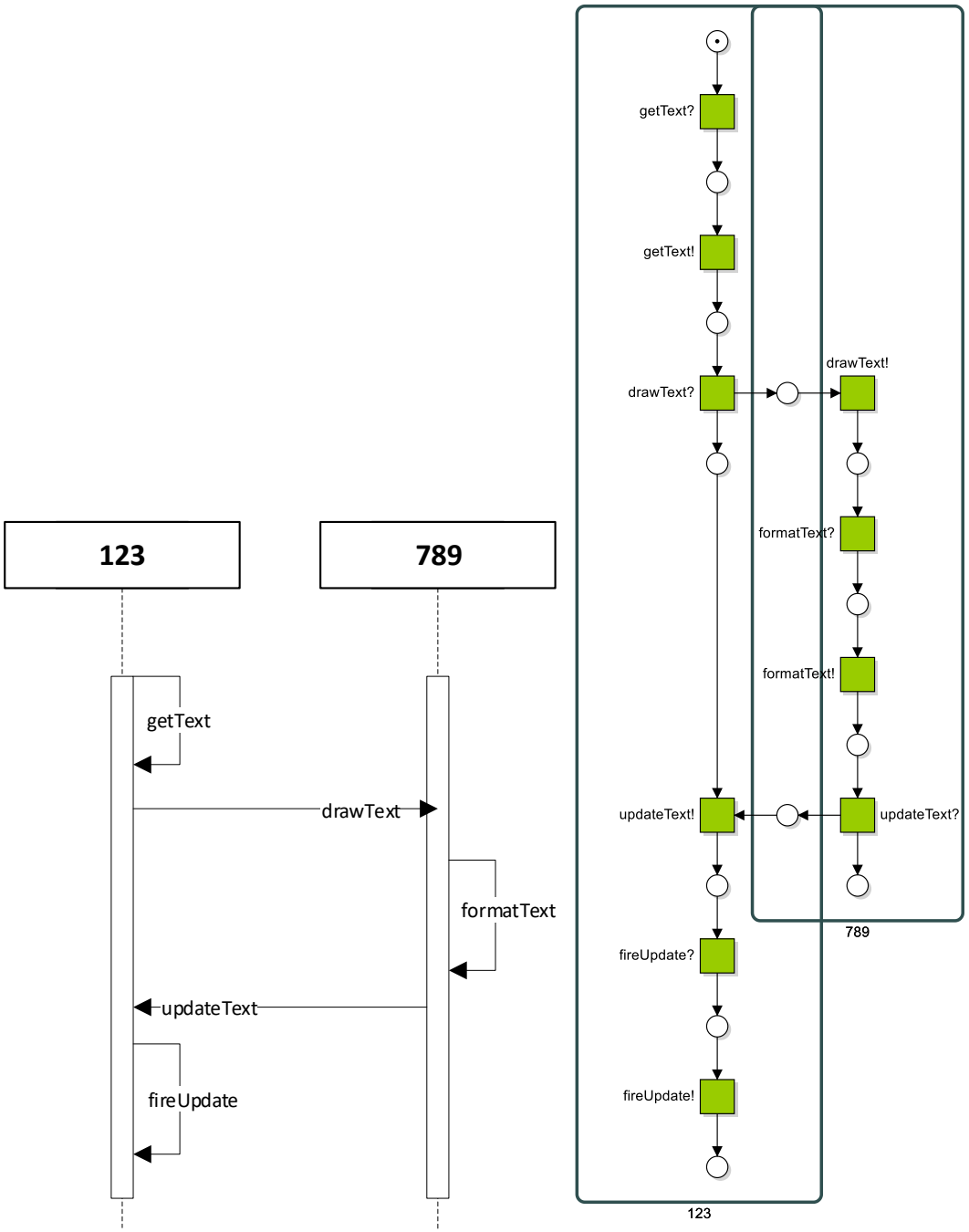


Figure 11: Message sequence diagram (left) and Petri net of com and org.Model.123

The original log can now be defined as the behavior of all its components:

$$\mathcal{B}(S, \{\langle org, Model, 123\rangle, \langle org, Model, 321\rangle, \langle org, Validator, 456\rangle, \langle com, ui, GUI, 789\rangle\}) = (\langle s_1, s_2, s_3, s_4, s_5, s_6, s_7\rangle) = S$$

5.4 COLLAPSIBLE CONTAINERS

As mentioned before, components in the HIM are *collapsible:* any container can be collapsed, so that all child components and their behavior is abstracted to the container. For example, if we

collapse the container $\langle org,\ Model\rangle$, we remove its children (and their children) from the component tree:

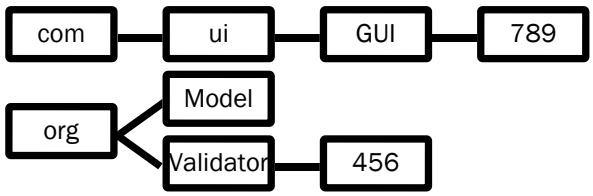


Figure 12: Running example component tree with org.Model collapsed

The child components of `Model`, `123` and `321`, are now abstracted into their parent, $\langle org,\ Model\rangle$. We define a rename function for renaming all the children of the collapsed container: ρ: $(\mathcal{O} \cup \mathcal{T}) \rightarrow \mathcal{T}$. Transforming the original log S to represent this, we get S':

ID	Caller	Callee	Message
1	org.Model	org.Model	getText
2	org.Model	org.Validator.456	validateText
3	org.Model	com.ui.GUI.789	drawText
4	com.ui.GUI.789	com.ui.GUI.789	formatText
5	com.ui.GUI.789	org.Model	updateText
6	org.Model	org.Model	fireUpdate
7	org.Model	org.Model	fireUpdate

Table 6: Running example log with org.Model collapsed

The behavior of the collapsed container is the same as it was before. Just as the behavior of *org.Model* ($\mathcal{B}(S, \langle org, Model\rangle)$) was ($\langle s_1, s_2, s_3, s_6, s_7\rangle$) in S, it is still ($\langle s_1, s_2, s_3, s_6, s_7\rangle$) in S'. The callers and callees in $S'$are however renamed: each sequence that started with $\langle org, Model\rangle$ is now truncated to just $\langle org, Model\rangle$. Collapsing $\langle org, Model\rangle$ doesn't change the behavior of our system, but it does change the way we represent it in process models and other diagrams. Because $\langle org, Model\rangle$ is now a leaf node in the component tree, we model its behavior as one entity whereas we considered its children ($\langle org, Model, 123\rangle$ and $\langle org, Model, 321\rangle$) separately.

The effect of is different for different formalisms. Looking at the log in Table 6, it is still a lot like the log before we collapsed a container. Similarly, only the interactions are different when the log is represented as a choreography model. In Figure 13 both the original and the collapsed log are displayed. The second model represent the collapsed log, with all differences highlighted in italic.

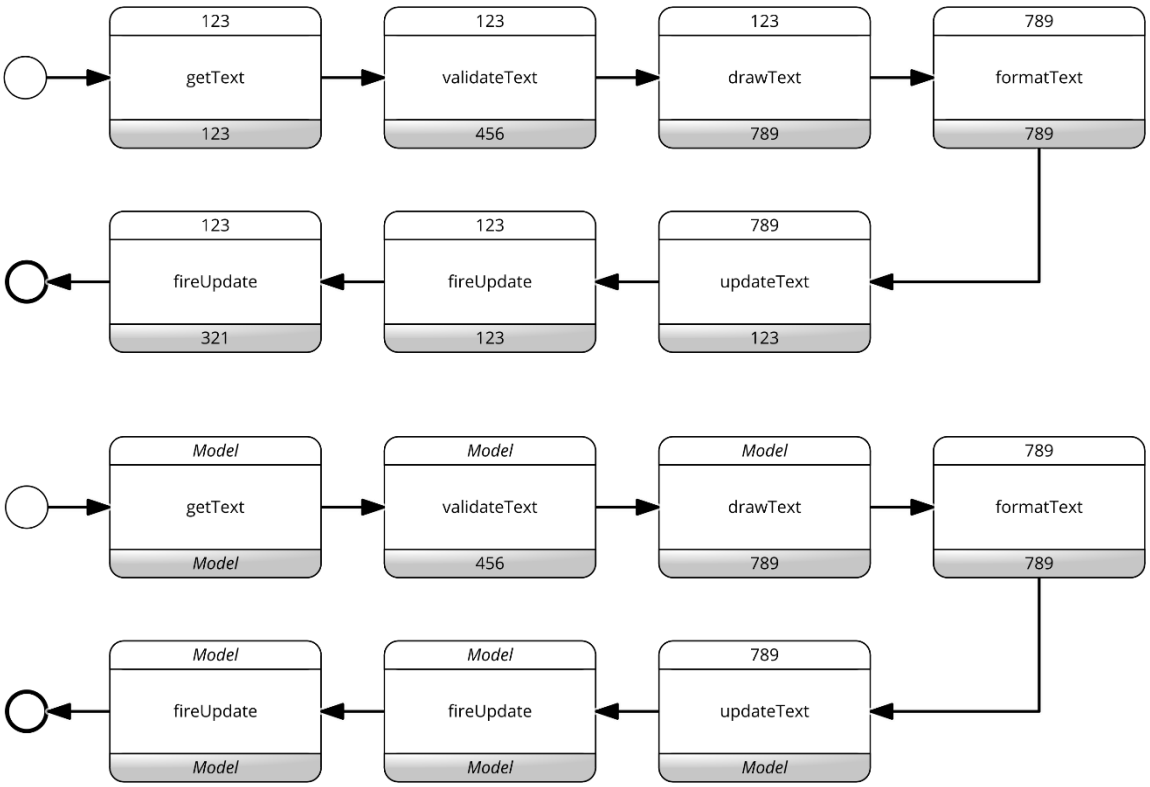


Figure 13: Two BPMN choreography models, representing the first and second running example logs, respectively

If we were to create a HIM in the same manner as Figure 8, we get a simpler and more compact HIM. Unlike the choreography models, the HIM changes not only in labeling but also in structure. Similarly, when representing the behavior of the two logs as Petri nets, we get structurally different Petri nets. This difference is displayed in Appendix A, Figure 26.

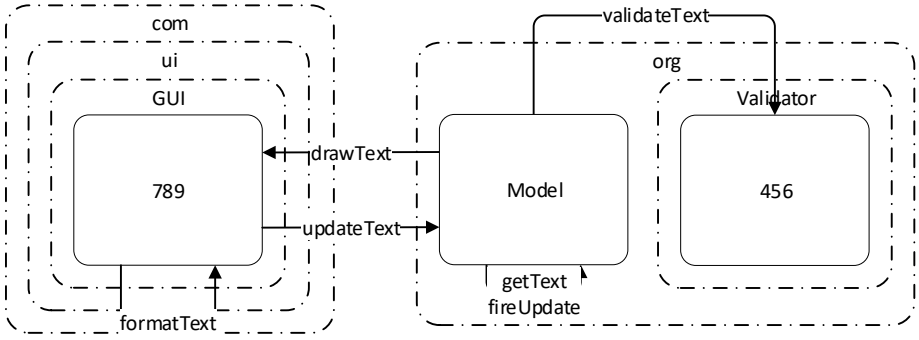


Figure 14: Running example HIM with org.Model collapsed

5.5 DISCOVERING PROCESSES

Hierarchical interaction models provide users with information about the hierarchical structure of software and the relationships between different components. On the other hand, HIMs do not give detailed insight into the behavior of software the same way process models do. This why for example in Figure 11 we represent the behavior of *com* as a Petri net. We therefore propose to use HIMs together with process models in a way similar as demonstrated in Section 5.3.

The basic approach to creating process models complementing HIMs is as that the logs that are used to create HIMs can also be used as inputs for process mining algorithms. While the HIM shows which components interact, it does not show in what order the interactions happen, or what the repeating patterns are within the interactions.

Most commonly used process mining algorithms take a single event attribute as the unique identifier of that event. In a business situation this would usually be an event name, such as 'check credit score' or 'send package'. Every event with the name 'send package' is then the same activity. More information on how process mining algorithms use events to abstract processes can be found in [15].

5.5.1 Extracting processes from software execution data

If we want process mining algorithms to correctly recognize processes within our logs, we need to carefully consider what our event identifiers should look like. The closest thing we have to an activity descriptor is the message. This however would not make a very good descriptor of the interaction at hand, because different classes could have entirely different methods with the same name or signature. We therefore need to introduce information about the class to which the method belongs, which in our case is the FQN of the callee object.

One major problem with software execution data is that there is no clear distinction between processes. When is a set of interactions a process? What constitutes a process as opposed to a process instance? There are several intuitive answers possible, for example:

1. An object lifecycle is a process instance, where the class lifecycle is a process
2. A request from call to response is a process instance, where the kind of request is a process
3. An API call from call to completion is a process instance, where the endpoint is a process
4. A run of an entire program or service is a process instance, where all runs of that program are a process
5. A session from start to finish is a process instance, where all sessions are a process
6. Processing a file from opening the file to closing it is a process instance, processing a certain kind of file is a process

Some of these suggestions are quite abstract, such as 1,4 and 5. Others, such as 2, 3 and 6 are only applicable to certain types of software and require more specific instrumentation than just recording all method calls. Nevertheless, they are all valid proposals that can be useful in certain contexts. For the purposes of this thesis, we want to create a method that can be applied to a wide variety of applications, while still creating compact (and therefore more easily analyzable) process models.

Our proposal is to use the behavior between two objects as a process instance, and the behavior of their containers as a process.

5.5.2 An example

ID	Caller	Message	Callee	Case
1	org.Model.123	setColor	com.ui.GUI.789	A
2	org.Model.123	setSize	com.ui.GUI.789	A
3	org.Model.123	drawText	com.ui.GUI.789	A
4	org.Model.321	setColor	com.ui.GUI.789	B
5	org.Model.321	setSize	com.ui.GUI.789	B
6	org.Model.321	showError	com.ui.GUI.789	B
7	org.Model.321	drawText	com.ui.GUI.789	B

Table 7: A second example log

Table 7 shows a second example log. Just like the first log it holds seven interactions involving three of the same objects. There are two objects from the `org.Model` container: `123` and `321`. They both exclusively have interactions with `com.ui.GUI.789`. The messages are identical, with one exception. First `123` sends `setColor` followed by `setSize` and finally `drawText`. `321` sends `setColor`, then `setSize`, then `showError` before coming to `drawText`.

We established that for our purposes the behavior between pairs of objects are process instances and the behavior between the containers of these objects as the process. The process here is between `org.Model` and `com.ui.GUI`. The process instances (commonly called cases in process mining) are between all `org.Model` objects (`123` and `321`) and all `com.ui.GUI` objects (only `789`). There are then two different object pairs that have interactions: $(org.Model.123, com.ui.GUI.789)$ and $(org.Model.321, com.ui.GUI.789)$. Consequently, we have two cases or process instances; the former is marked A in Table 7 and the latter is marked B.

A process is a generalization of all its process instances. To explain how this generalization is done, we give a simple explanation of what process algorithms do. We will not cover any process discovery algorithms in detail – their exact workings are quite different from each other and can be found in their respective literature.

Given a set of two process instances, our hypothetical algorithm will create a single process that covers both process instances. The process is modeled in a process model, which for our purposes is a Petri net. What we want the algorithm to do is recognize that interactions 1, 2 and 3 in Table 7 are the same as interactions 4, 5 and 7, and then conclude that interaction 6 is optional. Recall that most algorithms take a single attribute such as the event name as identifier. If the names of interactions 1, 2 and 3 were identical to interactions 4, 5 and 7, the algorithm should recognize the events as identical as well.

Because we don't develop our own process discovery algorithm, we must change the input of the algorithm to get the results we want. What we do is that we name each event after the caller, the message and the callee. This is the minimum information that makes an interaction unique and therefore identifies it.

ID	Event name	PI
1	org.Model.123 -> setColor -> com.ui.GUI.789	A
2	org.Model.123 -> setSize -> com.ui.GUI.789	A
3	org.Model.123 -> drawText -> com.ui.GUI.789	A
4	org.Model.321 -> setColor -> com.ui.GUI.789	B
5	org.Model.321 -> setSize -> com.ui.GUI.789	B
6	org.Model.321 -> showError -> com.ui.GUI.789	B
7	org.Model.321 -> drawText -> com.ui.GUI.789	B

Table 8: Log with unique event names

ID	Event name	PI
1	org.Model -> setColor -> com.ui.GUI	A
2	org.Model -> setSize -> com.ui.GUI	A
3	org.Model -> drawText -> com.ui.GUI	A
4	org.Model -> setColor -> com.ui.GUI	B
5	org.Model -> setSize -> com.ui.GUI	B
6	org.Model -> showError -> com.ui.GUI	B
7	org.Model -> drawText -> com.ui.GUI	B

Table 9: Log with abstracted event names

Table 8 shows what our example log looks like if we combine the caller, callee and message into one event name. If we were to feed this log into a process mining algorithm, we would get a process model where either all of process instance (PI) A or case B happened: to the algorithm, they are entirely different. As such, we remove all the object information and abstract the objects to their containers, matching our notion of process and process instance. The result is Table 9.

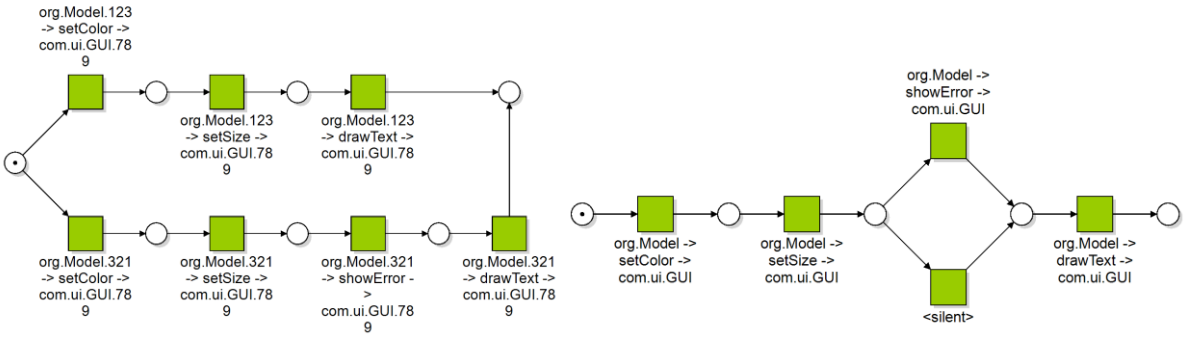


Figure 15: Petri net fitting the original event names and fitting the abstracted event names

The difference is evident in a process model. In Figure 15 we show a model that could be mined from Table 8 on the left and one from Table 9 on the right. The former has seven transitions, one for each interaction in the log, as they are unique to the algorithm. The latter has five transitions, of which one is silent. Just as we wanted, interaction 1 is seen as identical to 4, 2 as identical to 5 and 3 as identical to 7.

Interaction 6, org.Model -> showError -> com.ui.GUI, is optional. This makes sense given the input log, and results in a more compact and (to our intuition) more accurate model. To model this, we

created an 'exclusive or': either the optional transition is fired, or a 'silent' transition is fired. This silent transition, also called a τ step, is there for syntactic reasons and should be ignored for semantic purposes. Note that there are several possible Petri nets that fit the log in Table 9.

5.6 IMPLEMENTATION: INTERACTIONS PROM PACKAGE

The second toolset we created is the *Interaction* package for ProM[6]. This package contains three plugins: Interaction Builder and two variants of Interaction to XLog. Interaction Builder is a plugin that accepts an XES log and creates an interaction log. The user specifies which fields in the XES Log correspond to the caller, the callee and optionally the call and the plugin does the rest.

The two plugin Interaction to XLog variants both do the same thing: they take an interaction log and allow the user to select objects within that log. The plugin then creates a sub-log of the interaction log, containing all interactions between the selected objects. This sub-log is converted into an XES log similar to the XES log that was originally used as an input for the Interaction Builder plugin. The figure below illustrates the workflow for using the *Interaction* package.

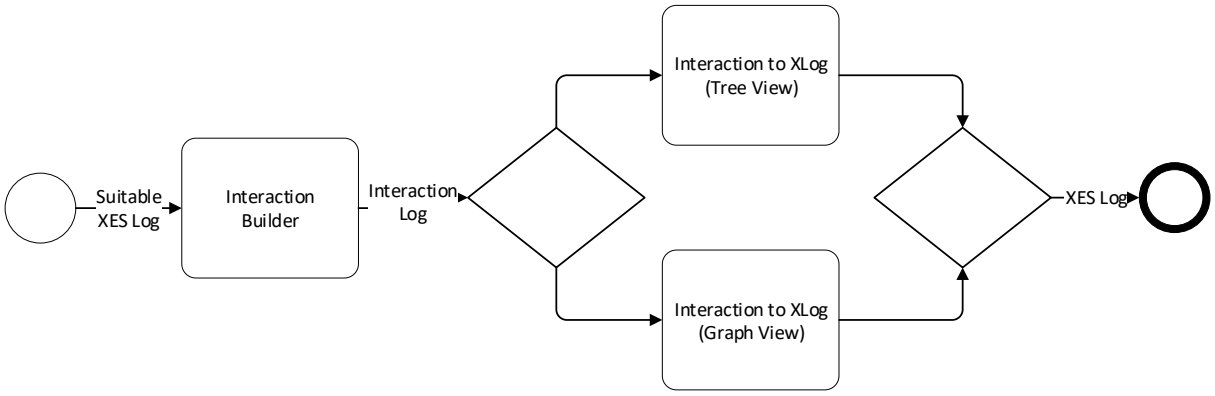


Figure 16: ProM workflow

Using the plugins, the user can select components that exist within the log for creating a 'sub-log'. The resulting XES log contains one case for every combination of two objects that fall within the selected components. These two objects do not have to be different objects: interactions between an object and itself are also included.

To illustrate what our plugins look like, we have included three screenshots below in Figure 17.

[6] https://github.com/ArchitectureMining/ProM-Interactions

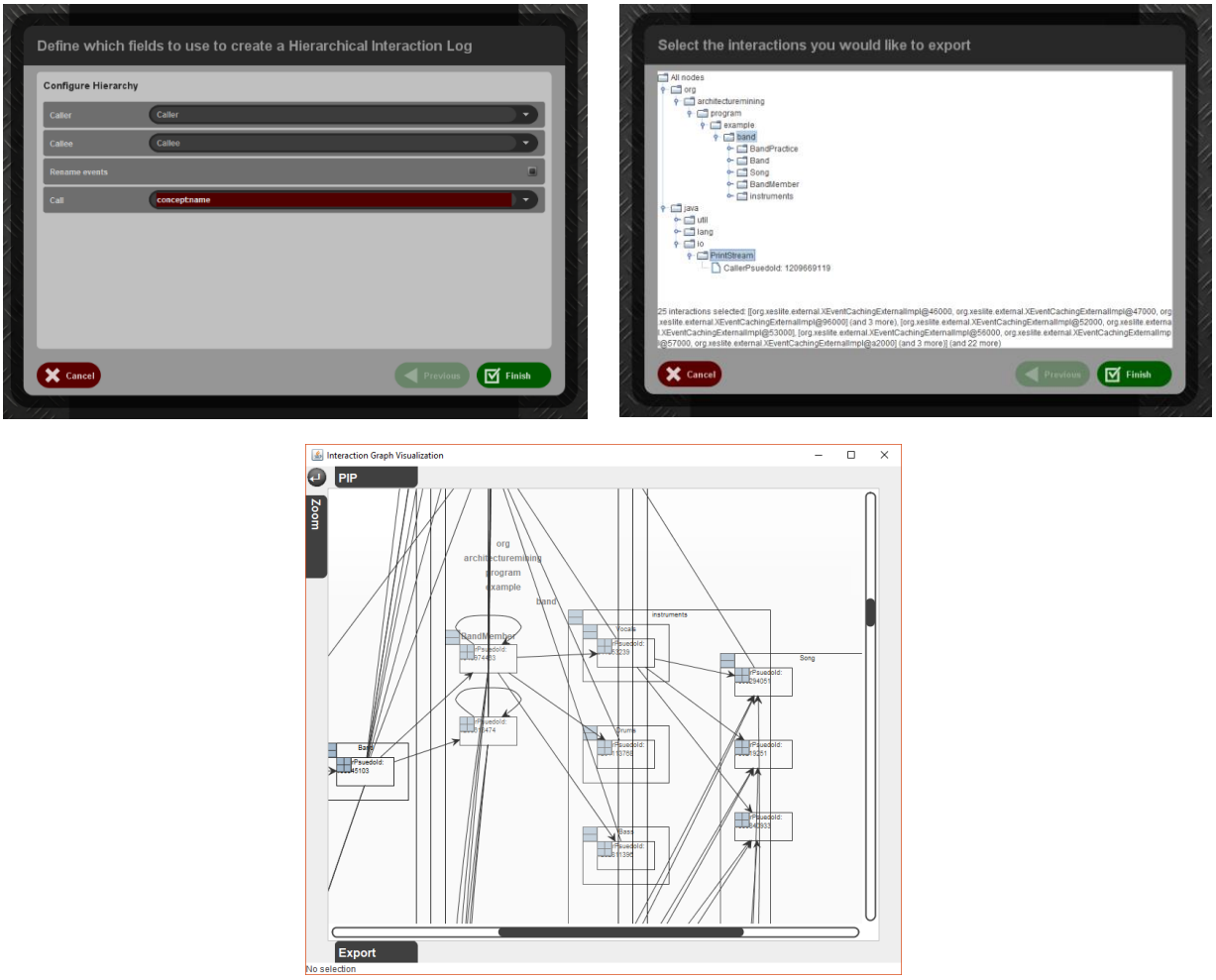


Figure 17: Screenshots of the Interaction Builder, Tree Visualization and Graph Visualization plugin (from top left, clockwise)

5.7 Conclusion

In this chapter, we answer the third sub-question, on how we can visualize software execution data in software architecture. Based on the logging technique presented in the previous chapter, we propose a visualization approach. Our visualization approach consists of two parts: the hierarchical architecture model (HIM) and process models created from these interactions.

The HIM is used for visualizing the components of a system, their structure and their interactions. It is based on functional architecture models and modified to better suit our purposes. Using several formal definitions, we explain the structure of the HIM as well as two operations: selecting interactions and collapsing containers. These two operations can be used to abstract interactions by hiding them and select interactions by choosing components.

While the HIM displays what interactions happen, it does not show any processes. A process can provide users with information about the order of interactions and is an abstraction of repeating patterns (cases) within the interaction log. While logs are the core input of both HIMs as well as process models, we need to do some modification to them for existing process mining algorithms to recognize them.

To make existing process mining algorithms recognize our logs, we must first find a mapping from our interactions to the process mining notion of a *case*. We choose to define a case (also called a process instance) as the interactions between any two given objects, and a process (their generalized form) as the interactions between their respective containers. Many other mappings are possible, but they are specific to certain use cases or more difficult to implement.

Now that we can split up our logs into different cases, we must convince process mining algorithms to see behavior of different objects within the same container as identical, given that they have the same message. To do so, we remove the object information and thereby abstract the behavior of objects to their containers. By combining the message, caller and callee into one field, we create a unique event identifier that process mining algorithms recognize.

Putting these two approaches together, we can give users an overview of the structure of the system that they can make more abstract and drill down upon. The drill down can be used to create detailed process diagrams with existing tooling, thereby visualizing behavior from software execution data.

6 METHOD OVERVIEW

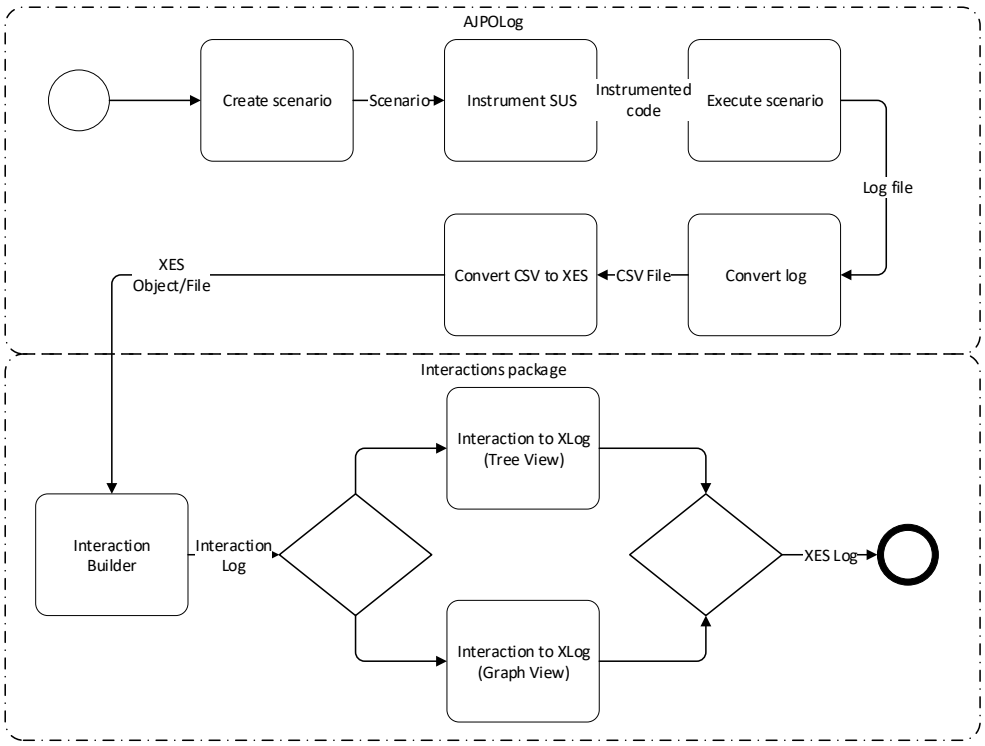


Figure 18: Full workflow of our SAR approach

Throughout our two case studies we adhere to a fixed method, show in Figure 18. This software architecture reconstruction (SAR) method combines the AJPOLog workflow (Figure 5) and the ProM workflow (Figure 16). It shows how we start by creating a scenario and finish with an XES log that can be used for mining process models. The final step of the AJPOLog workflow links directly into the first step of the Interactions package workflow, as can be seen in Figure 18.

For the two case studies in this thesis, we will evaluate our approach based on several aspects of the systems and our method. The relevant aspects are:

6.1 THE SYSTEM UNDER STUDY

We describe the system under study, what it is, how many lines of code it consists of, what it is written in and some information about the static architecture.

6.2 THE SCENARIO

The scenario defines not only how the reconstruction is executed, but also what the result will be. We therefore describe the scenario, giving an indication of what parts of the system will appear in the log and how this log can be recreated.

6.3 THE LOG

We give a short overview of the log, including how the raw and processed log relate to each other. With every case study we do a sanity check to make sure that the log and log conversion reflect our expectations.

6.4 VISUALIZATION

After describing the log, we use ProM to create visualizations of this log. We describe the results and whether they are useful in architecture reconstruction.

6.5 PROCESS ANALYSIS

Both visualization plugins we created can be used to filter logs and create smaller logs. These smaller logs are suitable for process mining. We check whether it matches our understanding of the software behavior on an intuitive level.

6.6 ARCHITECTURE CONFORMANCE

Finally, we compare a static architecture of the system under study to the results of our own architecture reconstruction approach. As mentioned in Section 3.1, using software execution data leads to different results than using source code as an input for SAR. By comparing a static architecture to our own dynamic architecture, we both check the accuracy of our approach and demonstrate the practical differences between static architecture reconstruction (using source code) and dynamic architecture reconstruction (using software execution data).

7 CASE STUDY 1: LAB SETTING

In this chapter, we will demonstrate and evaluate our approach on a simple program. We follow the method presented in the previous chapter and evaluate the different results our approach results.

7.1 SYSTEM UNDER STUDY

For our first case study, we created our a very simple system with few external factors affecting the results. The system is a useful test subject because:

- It is small enough to allow line-by-line analysis of the code
- It is written by us, so we know every detail of the intended architecture
- It doesn't allow for any user interaction, thus taking the limitations of scenarios out of the picture

There is no unused code in the program, and there is no user interaction possible. Running the program should therefore execute every line in the source code. This is an advantage when dealing with dynamic architecture reconstruction: we can be certain that all features of the program will appear in the log.

The *Band* package consists of eight classes, one of which is an interface. The following UML class diagram describes the static structure of the program. We leave out the package structure in this diagram. For reference, `Band`, `BandMember`, `BandPractice` and `Song` are in the package `org.architecturemining.program.example.band`, and the remaining classes are in `org.architecturemining.program.example.band.instruments`.

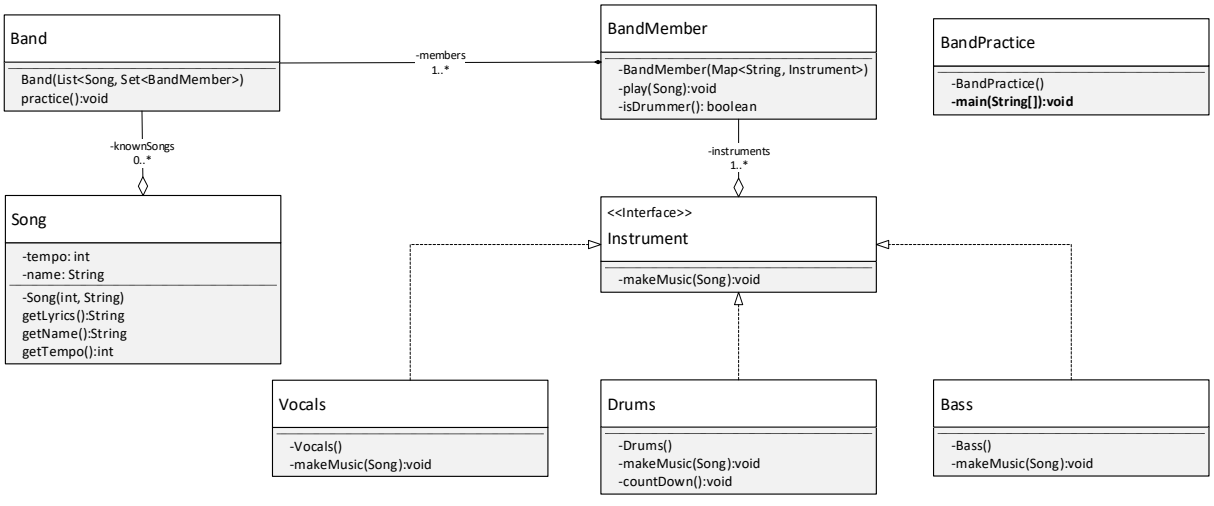


Figure 19: UML Class diagram of the test example

The entire program is around 250 lines of Java code. It has no external dependencies before being instrumented.

7.2 SCENARIO

There is no user interaction possible in this program, meaning there is no need for a scenario. We start the program and after a fixed time it terminates, having executed all code in the project.

7.3 LOG

The raw log from our example contains 256 entries. As expected, the processed CSV file has 129 lines: one header row and 128 entries: exactly $\frac{(method\ entries+method\ exits)}{2}$. According to this sanity check, no major problems exist with the log.

After creating our CSV, we used the default CSV to XES plugin in ProM to convert our resulting CSV file to an XES file. We use 'Thread' as a case column, 'Message' as an event column, and set the 'Start Time' and 'End Time' as start time and completion time. To make sure there are no problems with the input we tell ProM to stop on errors. In our case study the conversion went without problems. We select XESLite (MapDB with cache) as format.

7.4 VISUALIZATION

In Figure 20 the output of the interaction graph visualization plugin is shown. What is immediately obvious is the unusual shape of the diagram, and the poor use of the available space. A larger (and cropped) version of the diagram is available in Appendix A as Figure 27.

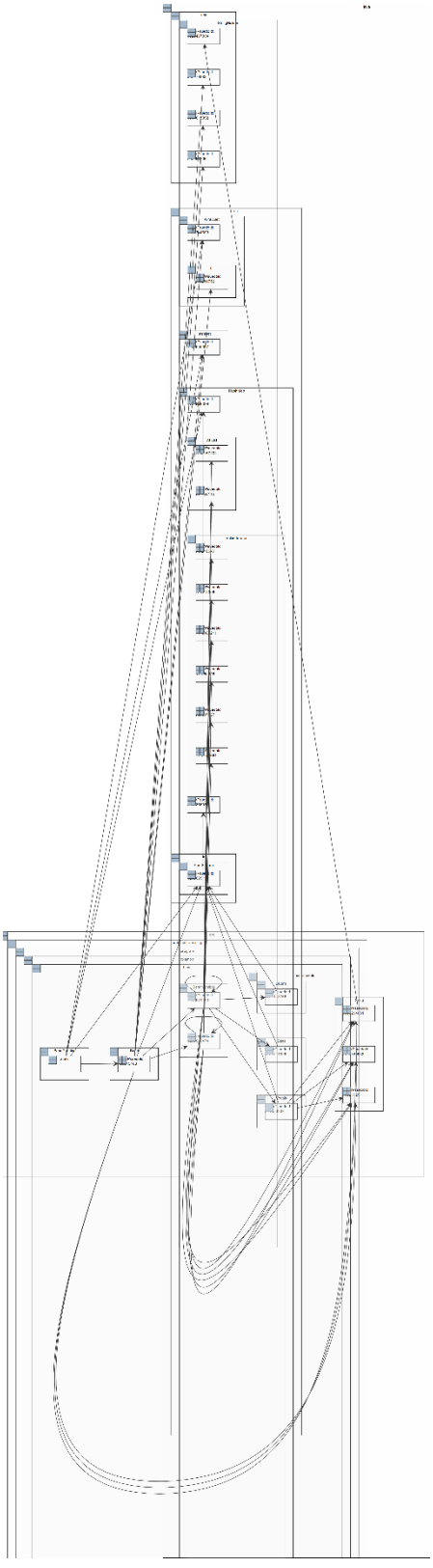


Figure 20: Uncropped ProM diagram output of the band example

Although the chaotic nature of the diagram make it difficult for us to analyse what exactly is being depicted, we can start by looking at the area in the middle. This is where we find the contents of the org.architecturemining package. A larger figure containing only this part is depicted as Figure 28.

We can see that there is a call from the `BandPractice` class to an instance of `Band`. `Band` then calls two `BandMember` objects, and each of these bandmember objects calls itself. `Band`, the `BandMember` objects and the `Vocals` object all call each of the `Song` objects.

We also tried the tree visualization plugin. It worked exactly as expected, creating an interactive tree view of all classes, packages and objects. We used this view to select the `BandMember` and `HashMap.ValueIterator` (`ValueIterator` is an inner class of `HashSet`) classes for the process analysis in the following section.

7.5 PROCESS ANALYSIS

We use the ProM Interactions package we developed to select a small subset of the objects in the log and create a smaller XES log. As an example, we chose the `BandMember` and `HashMap.ValueIterator` classes. Selecting these two classes creates eight cases, all between different instances of `ValueIterator` and `BandMember`.

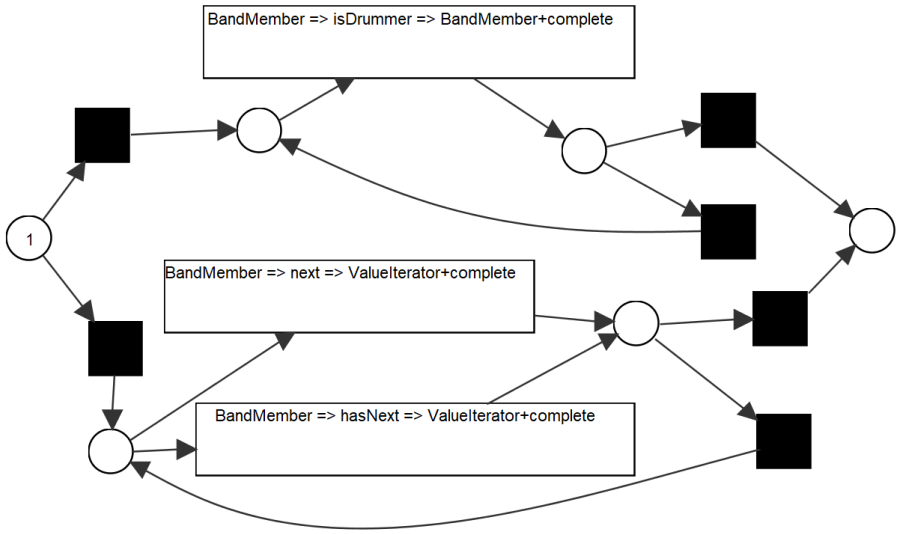


Figure 21: Petri net mined from BandMember and ValueIterator classes

Figure 21 is a Petri net mined from the sub-log we described above. It was mined with "Inductive Miner – Life Cycle (IMlc)" using the "Mine Petri net with Inductive Miner" plugin. This algorithm is explained in [33]. We used both the concept name (the event name, the part before the plus sign in the transition name) and the lifecycle transition (whether the event is a start or complete event, the part after the plus sign) as event classifier. The leftmost place with no incoming edges is the starting position; in the initial marking this contains one transition, allowing the first transitions to fire.

The resulting Petri net does match the processes in the original program. The mining algorithm mined the two different processes as choices within one process as a process model can only contain a single process. It seems that our pre-processing as described in Section 5.5 works correctly: eight cases were abstracted into two distinct flows within our process model. Like in the log we used as input for Inductive Miner, `isDrummer` is always followed by either the end of the log or itself. The second path, consisting of interactions with `ValueIterator`, matches the logs in that `hasNext` is either followed by next or the final event in a loop.

7.6 ARCHITECTURE CONFORMANCE

Our example program is very small, leading to a small log with 'only' 128 method calls in it. Because this log is relatively small, we can analyze whether every call we expect to see based on our code is also in the log. That way, we can check if our instrumentation method works as expected.

To do so, we first create an intended dynamic architecture, so that we can compare our mined architecture to it. We created this architecture by manually going through every line of code and following these rules:

1. For every .java file in the project, add the classes in that file as a node
2. If there is an explicit call to a method (of the form `<object>.<method name>(<arguments>)`, draw an edge from the current class to the method's class
    a. If the method being called is outside of the project, add that class as a node as well
    b. Label the edge with the name of the method and the types of its arguments
3. If there is an implicit call to a constructor (such as when the `new` keyword is used), add that as a method call to a method of the same name as the class
4. Create a group for every package and label it

The result can be compared to a call graph. Please note that `System.out` is field in the System class, meaning a call to `System.out.println()` is a call to `PrintStream`. For this reason we included `println()` calls as calls to `PrintStream`. These rules resulted in the model in Figure 22. The dotted lines indicate a group, or package boundaries. We simplified the package structure by putting all standard Java classes in a single group.

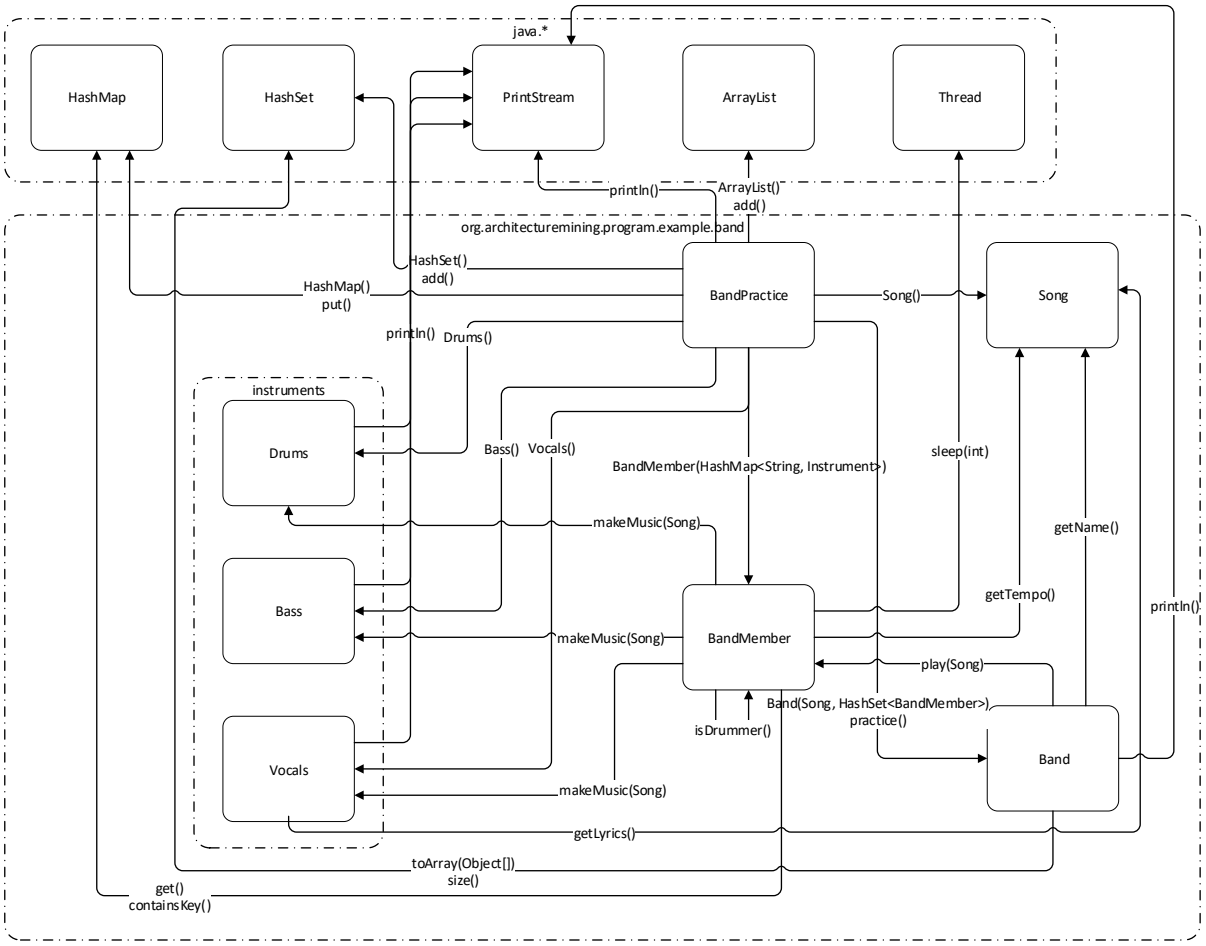


Figure 22: Expected realized architecture outcome of the test example

Caller	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
BandPractice		I	RI	I	I	I	I	RI	RI	RI	RI					
BandMember		RI		RI	RI	RI	RI	RI				I			R	R
Band		RI		RI					RI	RI	R		R	R		
Song														R		
Drums										RI						
Bass										RI						
Vocals				RI						RI						

COLUMN	NAME
1	BandPractice
2	BandMember
3	Band
4	Song
5	Drums
6	Bass
7	Vocals
8	HashMap
9	HashSet
10	PrintStream
11	ArrayList
12	Thread
13	ArrayList.Itr
14	StringBuilder
15	HashMap.Values
16	HashMap.ValueIterator

Table 10: Band Example Intended Architecture vs Realized Architecture

In Table 10, we compare the intended architecture of the band example to the realized architecture. The relations we found in the comparison are divided into three kinds: the relation exists in the intended architecture only, it exists in the realized architecture only, or the relation exists in both. In Table 10 these are indicated as I, R, or RI respectively. The intended architecture contains a total of 25 relations, the same as the implemented architecture. In 19 of those 25 relations the realized architecture matched the intended architecture, marked in green in the table.

Given those numbers, we know that 6 relations were in the intended architecture but not featured in the realized architecture. Coincidentally, the reverse is also true: 6 relations that were in the realized architecture were not in the intended architecture.

For example, in our rules we did not take into account the syntactic sugar involved in `for each` calls (like `for(item : items)`). Calls to `ArrayList.Itr`, `StringBuilder`, `HashMap.Values` and `HashMap.ValueIterator` were not present in the intended architecture shown in Figure

22. These classes are not explicitly referenced in the source code and based on their occurrence and naming we assume that they are related to our `for` loops.

While the `ArrayList` class did occur in the intended architecture, it was not being called by the `Band` class. Analysis of the code in the `Band` class revealed that there too there was a use of a for each loop.

The relations that were included in the intended architecture but not in the realized architecture can be divided into two groups. 5 of the 6 relations are calls from the `BandPractice` class. The remaining relation is a call from `BandMember` to thread. The discrepancies related to `BandPractice` all seem to be situations in which methods are called and one of the arguments contains the `new` keyword. For example:

```
aliceInstruments.put("Drums", new Drums());
```

A possibility is that the actual call to the constructor of `Drums` happens in the `HashMap` class, which is outside of what AspectJ can instrument. Consequently, there is no call to the `Drums` class from `BandPractice`.

Statistically speaking, we have 19 true positives in our relationship comparison, 6 false negatives and 6 false positives. We do not consider the true negatives, as this number is arbitrary: the number of relationships that do not exist and are not being detected is practically infinite. The precision of our method ($TP/(TP+FP)$) is 0.76. The recall ($TP/(TP+FN)$) is also 0.76, as there are 6 false positives and 6 false negatives. Succinctly, we can state that in our toy example about 3 in 4 of the calls we expected to see were recognized. It is worth noting that our own ignorance of the Java language's syntax played a large part in this number.

7.7 Conclusion

In this chapter, we demonstrated our approach on a very small program. To test our ProM plugins, we tried both the graph visualization as well as the tree visualization. The tree visualization worked well and without any notable issues. The graph visualization nevertheless suffered from a poor layout, making it harder to use than we expected.

Using the tree visualization plugin, we created a small log to test the applicability of process mining tools to our software logs. Using inductive miner the resulting process model matches what we expected and gives a correct representation of the behavior in the log.

We successfully used our plugins to select interactions and convert them to an XES log. From this XES log we were able to mine a correct process model that matched our expectations. We found that at a method-by-method level, the method calls AJPOLog registered mostly matched what we expected. Nevertheless, some calls were not as we expected, mostly because we were not familiar with the way that the Java method call syntax relates to its semantics.

8 CASE STUDY 2: JABREF

8.1 SYSTEM UNDER STUDY

The second program we analyzed is JabRef, an open source bibliography reference manager [34]. It is written in Java, and as such can be instrumented using AJPOLog. Whereas our test application (the band example) did not support user interaction, JabRef is a fully-featured desktop application with a GUI. This complicates the process of instrumentation, because we need to further specify *what* we want to model. We can only capture the features of a program that we use, so we need to adjust our scenario accordingly.

The choice for JabRef was in part motivated by the existing research on its architecture. Olsson et al. [35] studied JabRef 3.7 in their research, and made both the results as well as the source code they used available online. This version of JabRef has a documented architecture created by the developers [36], giving us an intended architecture to compare our reconstructed architecture to. For these reasons, we decided to research version 3.7 instead of the latest version of JabRef, which at the time of writing is 4.3.1.

JabRef is described by Olsson et al. as a 'medium-sized' system [35]. Version 3.7 consists of 1,016 Java files containing 1,537 classes for a total of 88,617 lines of code. This means it is significantly larger than our previous example: around 350x the lines of code and around 200x the number of classes.

In the remainder of this chapter, we will go through the SAR process described in Figure 18.

8.2 SCENARIO

We created a scenario that used some of the basic functionalities of JabRef. The scenario we created is simple, only using functionality that we expect to be commonly used. A more extensive scenario would give a more complete view of JabRef. But more data means slower processing, more edge cases to consider and more reconstructed architecture to analyze. We settled on the following scenario:

1. Open JabRef
2. Create a new database
3. Create a new entry
4. Select "Article"
5. Enter an author, title and key
6. Save the database
7. Export the database (File > Export)
8. Close JabRef

8.3 LOG

We executed the above scenario in our instrumented version of JabRef. The resulting log file is around 140 megabytes in size, containing a total of 629,831 lines. Next, we used the conversion script included in AJPOLog to convert our newly created log into a CSV file. While converting the

file, the script detected a method call that did not exit and automatically fixed the call. The following information was provided about the call:

Callee	net.sf.jabref.logic.remote.server.RemoteListenerServer.CallerPsuedoId: 240576935
Caller	net.sf.jabref.logic.remote.server.RemoteListenerServerThread.CallerPsuedoId: 409021659
Type	Entry
Timestamp	2018-11-30T12:24:44,431
Message	public void net.sf.jabref.logic.remote.server.RemoteListenerServer.run()
Thread	[JabRef - Remote Listener Server on port 6050]

Processing the entire file takes about 13 seconds on a laptop equipped with an Intel i3 3120m processor. More modern and powerful processors should be able to do this much faster. This part of the approach should scale well to larger logs and scenarios.

In theory, the processed file should have about half the lines of the original file because every pair entry and exit events is combined. One line is added as a header row so that ProM has a label for every column in the CSV file. This calculation proved correct for the band example in the previous chapter.

However, the processed file from our JabRef scenario has 314,606 lines, which is 311 fewer than what we expected. We expected 314,917 lines, because we know that one line is missing in the original file, making 629,832 lines. Every line in the new file should correspond to exactly 2 in the original, halving that number to 314,916. Finally, we add a header line, bringing the count to 314,917. We would have accepted one extra or missing line in case either the original or the new file had an empty last line added to it.

To find what went wrong, we modified the code in the conversion script. Originally, it only processed lines that had ‘Entry’ or ‘Exit’ in the place we expected. Our modification was such that any line that did not contain either of those values would be written to a file and found 622 log lines that were being skipped by the script. Because we expected 311 more lines in our log with the logic that every 2 lines in the original make 1 line in the CSV, we have found all our missing lines.

Looking at the contents of the ‘missing’ lines, it seems that our script was right in not converting these lines. Apparently, JabRef also uses Log4j as a logging mechanism, and the missing lines were debug messages generated by JabRef itself. This is a shortcoming of AJPOLog: it does not consider any Log4j usage by the system under study and consequently intertwines its own logs with that of the SUS. Despite this flaw, our CSV should still be fit for usage if none of the logs created by JabRef had ‘Entry’ or ‘Exit’ as its log message.

We used the same process for converting CSV to XES as in Section 7.3. The resulting .xes file is around 300 MB in size.

In our Interaction Miner plugin, we mapped the Caller and Callee to their eponymous fields. We enable the ‘rename’ field and use ‘concept:name’ (originally ‘Message’ in the CSV) as the Call field.

8.4 VISUALIZATION

We first tried the graph visualization plugin we created on JabRef. Despite this, it seems that JGraph does not perform well enough for such a large graph. After approximately four hours of waiting, the plugin was still not giving any graphical output. We terminated the process as it seemed we would not be getting useable results within an acceptable amount of time.

Conversely, the tree visualization gave results within seconds.

8.5 PROCESS ANALYSIS

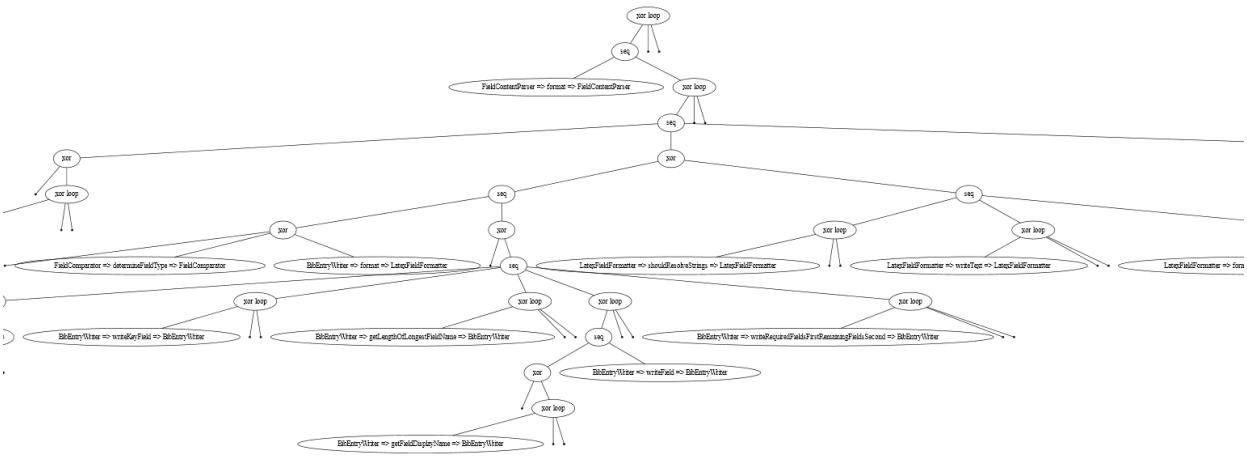


Figure 23: Part of the process tree created from the JabRef interaction log

In Figure 23 a section of a process tree generated from our interaction log is displayed. A bigger version is available in Appendix A as Figure 29. The interaction graph used as input is a sub-log of the overall interaction graph: it only involves interactions of the `net.sf.jabref.logic.bibtex` package with itself. We used the interaction Interaction to XLog (Tree view) plugin to create this sub-log of the JabRef 3.7 scenario interaction log.

This process tree was generated using the inductive miner [37]. We used the Graphviz visualization in the latest version to create an SVG format diagram.

8.6 ARCHITECTURE CONFORMANCE

In the case of JabRef, it is not possible to do a line-by-line comparison like we did with the band example. The amount of time required to go through every line of code would be far too large and comparing the logs to the code would take a similarly long time. Despite this, it is possible to compare the static, intended architecture to the results of our logs as the JabRef developers have made an architecture available[7].

This architecture is described as high-level documentation and describes JabRef as consisting of several packages, each of which has a set of permitted dependencies to other packages. The permitted dependencies are shown in Figure 24. Each edge indicates that a dependency (and therefore an interaction) between those packages is allowed.

[7] https://github.com/JabRef/jabref/wiki/High-Level-Documentation

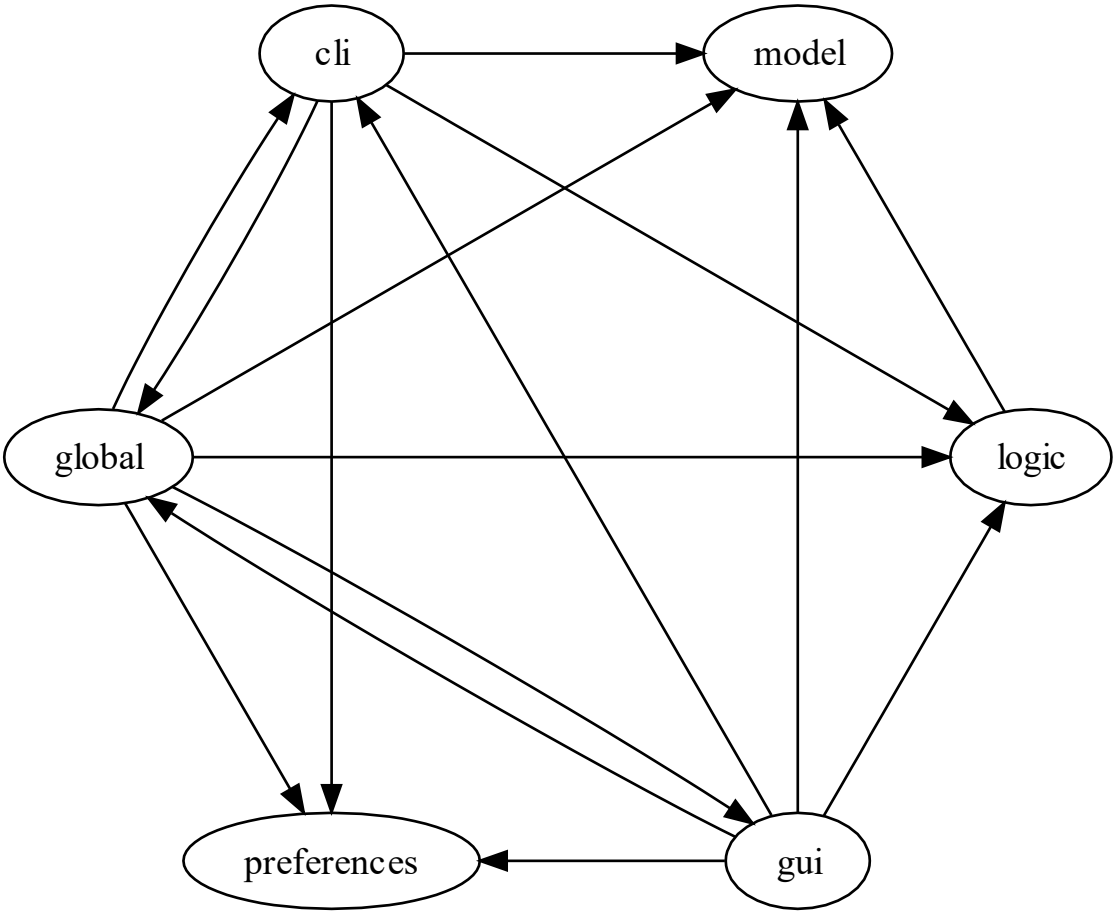


Figure 24: Permitted dependencies in the JabRef intended architecture

To compare the intended architecture to the implemented architecture, we created a Python script to read the CSV log of our JabRef 3.7 scenario to create a graph of all interactions. This script is included in AJPOLog[8]. In JabRef 3.7 the structure of the Java packages does not match the structure described in the documentation, meaning a mapping is required to match objects in the log to packages in the intended architecture. The script applies a pre-defined mapping to logs, converting each entry to a caller and callee pair. Every pair is then a directed edge, with the caller as the tail and the callee as the head. The edges together form a directed graph, just like the one in Figure 24.

JabRef 3.7 has previously been subject to research on architecture conformance during SAEroCON 2016 [38]. Leo Pruijt and Tobias Dietz together analyzed the implemented architecture of JabRef and researched its conformance to the intended architecture [36]. They used HUSACCT, a tool for conformance analysis of static architectures based on source code. For their analysis they created a mapping of packages to the packages in the intended architecture. A modified version of this mapping can be found in the Appendix of this thesis as Table 14. We modified the mapping slightly to suit our application better.

Using the mapping by Pruijt and Dietz and our log-to-graph script we created the graph in Figure 25. The packages are slightly different than the ones in the intended architecture: the mapping

[8] https://github.com/tijmendj/AJPOLog

uses uppercase letters and includes the external libraries Swing-AWT, Java SQL and Oracle SQL. The package ‘cli’ is now called ‘CommandLineInterface’.

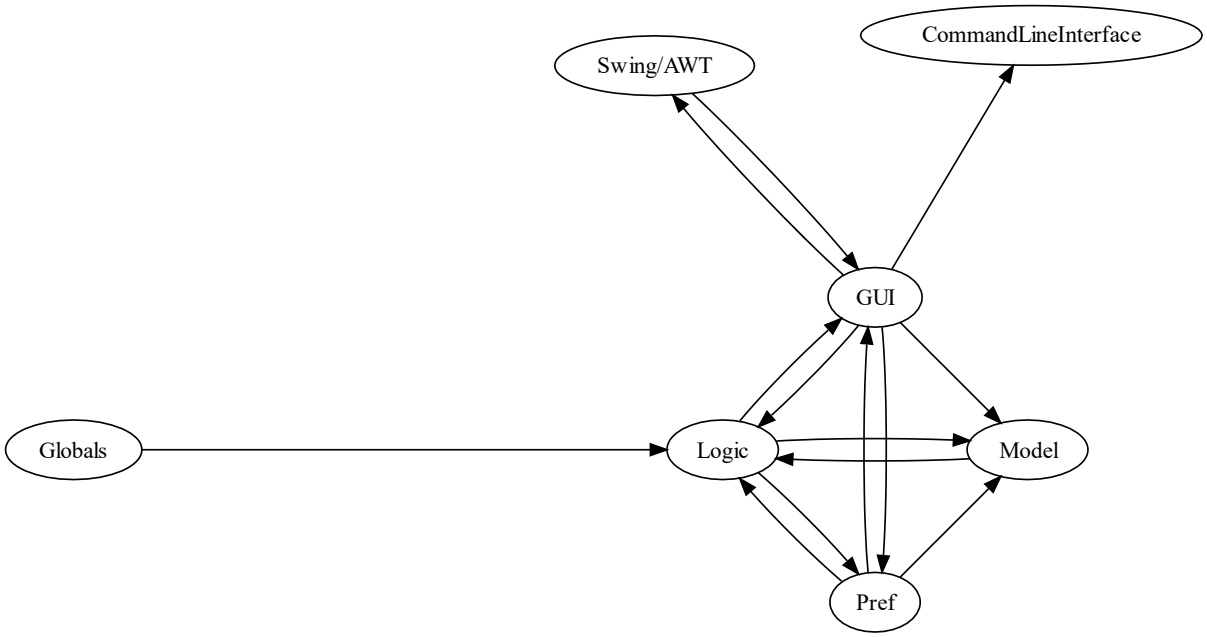


Figure 25: Realized architecture with Tobias Dietz's mapping [36]

First, we compare the architecture we reconstructed using AJPOLog to the rules intended architecture. In Table 11 the Caller and Callee column represent the caller-callee pairs we found in our log once we mapped every package according to the mapping in Table 14. The final column indicates whether these interactions are allowed according to the intended architecture. We find that of twelve interactions, only six were allowed in the intended architecture. The remaining six interactions are violations of the intended architecture.

Caller	Callee	Allowed?
GUI	CommandLineInterface	Yes
GUI	Model	Yes
GUI	Pref	Yes
GUI	Logic	Yes
Globals	Logic	Yes
Pref	GUI	No
Pref	Logic	No
Pref	Model	No
Model	Logic	No
Logic	GUI	No
Logic	Pref	No
Logic	Model	Yes

Table 11: JabRef 3.7 implemented architecture versus intended architecture

Second, we can check if our dynamic approach found the same violations as a static approach. As discussed, Pruijt and Dietz previously used HUSACCT to do static conformance checking on JabRef. The results of their analysis are also included in the proceedings of SAEroCON 2016 [36].

To demonstrate the difference between a static and dynamic architecture conformance checking approach, we compare the results of Pruijt and Dietz's analysis to ours. To make the comparison fair, we use the rules as formalized in the SAEroCON analysis instead of the ones derived from the intended architecture. The rules that were violated in the implemented architecture according to Pruijt and Dietz ([36]) are shown in Table 12.

Id	*Logical module from*	*Rule type*	*Logical module to*	*Violated in AJPOLOG?*
1	GUI	Is not allowed to use	CommandLineInterface	Yes
2	GUI	Is the only module allowed to use	External.Swing-AWT	No
3	Logic	Is the only module allowed to use	External.JavaSqlAndOracle	No
4	Logic	Is not allowed to back call		Yes
5	Logic	Is not allowed to use	Pref	Yes
6	Model	Is not allowed to back call		Yes
7	Model	Is not allowed to use	Pref	No
8	Pref	Is not allowed to use	GUI	Yes
9	Pref	Is not allowed to use	Globals	No

Table 12: Rules as used in [36] *and whether these rules are violated in our AJPOLog analysis*

A 'back call' in this instance is a call to a 'higher-level layer' [39, p. 30]. We re-interpret the intended architecture to fit this description, as it is not explicitly layered. Because *model* may only receive calls (see Figure 24), it is the lowest layer in the purported layered architecture. Likewise, *logic* is only allowed to call model and receives calls from other layers: it is therefore the second lowest layer. The remaining packages are not involved in a 'back call' rule, so they can be seen as a single layer above *logic*.

The final column in Table 12 indicates whether that violation found with HUSACCT was also found by us using AJPOLog. Five out of nine violations were found by both systems, and four out of five were only found using HUSACCT. Rule-by-rule, we found the following commonalities and differences:

1. The first violation is that GUI is not allowed to use CommandLineInterface. The intended architecture explicitly states that this is allowed. It seems this rule was created ad hoc at the time and has not been integrated in the intended architecture afterwards. While not considered a violation originally, AJPOLog did detect interactions that violated this rule.
2. The second violation is that only GUI should be connected to Swing/AWT. We did not find this violation in our model. Further analysis of the results in [36] shows that this violation occurs when setting a color in the preferences of JabRef, something that does not happen in the scenario.
3. This violation is between Logic and the SQL libraries. Our scenario did not include any usage of SQL functionality, so we could not find this violation.
4. The fourth violation is related to 'back calls'. In this context this means that there should not be any calls from Logic to any other module than Model. We can see that this rule is violated in our results as well.
5. The fifth rule is that Logic is not allowed to use Pref. This violation is in our model as well.

6. Like the fourth violated rule, model is not allowed to do back calls. Whereas Logic is only allowed to call Model, Model is not allowed to call any modules. This violation appears in our model just like in Pruijt and Dietz's analysis.
7. The seventh violation is that Model is not allowed to use Pref. This violation does not occur in our model. All violations HUSACCT found involve the DBMSConnectionProperties class, indicating that they are related to the database functionality of JabRef. Like with violation 3, our log contains no database functionality at all.
8. The eight violation is that Pref is not allowed to use GUI. This occurs in both our model as well as that of Pruijt and Dietz.
9. The ninth and final violation is that Pref is not allowed to use Globals. This violation does not occur in our model. All violations found using HUSACCT by Pruijt and Dietz are either import or access dependencies, which AJPOLog does not detect.

Moreover, there are two violations in our model that are not in Pruijt and Dietz's analysis. We filtered our log to find what these calls were, and we found a single call from Pref to Model and twelve calls that went from Pref to Logic. These calls are almost all in the same method of the same class and occur only once in the log. Of the violations found by our system but not by HUSACCT,
`jabref/src/main/java/net/sf/jabref/preferences/JabRefPreferences.java` line 1335 through 1344 account for 11 of 13 total calls that were not allowed. The remaining two are from line 1352 and 1353 of the same file.

8.6.1 Static versus dynamic architecture reconstruction in practice

The fundamental difference between HUSACCT and AJPOLog as methods to reconstruct an architecture is that HUSACCT uses static while AJPOLog uses dynamic data. A static approach should be able to detect all calls that occur in the code but will not be able to register any interactions that cannot be determined from the source code. This includes reflection and subtyping. Conversely, a dynamic approach can find interactions that are not in the source code but cannot find any interactions that did not occur when generating a log.

An example of the latter is the SQL functionality in JabRef. This functionality is not triggered in our scenario, meaning it does not show up in the logs. Consequently, it is not possible to reconstruct any of that functionality with a dynamic approach. Theoretically, a scenario that covers all functionality of a program would eliminate this issue. The band example in the first case study is a demonstration of such a scenario, as the system under study without any possible user interaction will always result in a log that is an exhaustive list of all possible interactions. Conversely, creating a scenario for JabRef that includes every interaction possible and 100% code coverage is infeasible.

One thing specific to HUSACCT and AJPOLog is that AJPOLog is designed to detect calls, while HUSACCT can detect several other dependency types. [39] lists several dependency types detected by static tools. Of those dependencies, AJPOLog will not detect import, declaration, access, inheritance and annotation dependencies. Indirect dependencies, regardless of their type will not be detected either. We found one example in which this created a practical difference: the ninth violation in Table 12, which is the only violation that does not involve any method calls.

AJPOLog is therefore not able to recognize it, even if the scenario covered the code in which the violations occurred.

What we cannot explain by the difference between static and dynamic reconstruction methods is the violations we found that were not found by HUSACCT. As we showed, the violations can be trivially found in the source code. Eclipse can resolve the call to the correct class as well, indicating that static analysis is not the problem.

8.7 Conclusion

In this chapter, we applied our software architecture reconstruction method to JabRef, an open source reference management tool. Unlike our example program, JabRef allows for user interaction, meaning we must first create a scenario. We created a short scenario covering only basic functionality. Even for such a short scenario, the resulting log was quite large: over 140 megabytes and containing over 300,000 method calls.

In fact, we had a few more log entries than we intended, as AJPOLog was conflicting with JabRef's own logging system. Luckily, our log processing script removed the log entries that we did not intend to be there automatically. When trying our visualization plugins, we found that the tree visualization worked fine. Nonetheless, the graph visualization plugin did not scale well to the size of our log. We did not get any result within a reasonable time, meaning we could not create a graph visualization.

As the tree visualization plugin still worked fine, we used it to create a sub-log like we did in the previous case study. We used inductive miner once again, this time generating a process tree instead of a Petri net. The model matched the behavior we expected from the sub-log.

We also evaluated our method by using AJPOLog for architecture conformance checking. Because the version of JabRef we used has been previously used in architecture conformance research, we could compare our results to those of a static architecture conformance checking approach. Using a mapping of Java packages to architectural elements created by Tobias Dietz and Leo Pruijt, we were able to reconstruct the implemented architecture of JabRef. We compared the implemented architecture to the intended architecture provided in the JabRef documentation and found that six of twelve interactions between elements were in violation of the intended architecture.

Comparing our results to a static analysis done using HUSACCT, we found that the theoretical differences between dynamic inputs and static inputs made a considerable difference in practice. We compared the nine architecture violations Pruijt and Dietz found using HUSACCT and found that our analysis picked up on five of these violations. Conversely, our analysis found one violation that HUSACCT did not pick up. This seems to confirm the idea that static and dynamic approaches complement each other.

The main theoretical difference that was demonstrated in practice is the coverage that static approaches provide versus dynamic approaches. HUSACCT uses the source code of a system to detect interactions, thereby recognizing (almost) all possible interactions that can be derived statically. Our dynamic approach, AJPOLog, only picks up on interactions that occur when the

scenario is executed. This is for example why our approach did not detect a violation where the *Logic* package interacted with a SQL database: our scenario did not use the database functionality of JabRef.

A more minor technical difference is that HUSACCT can detect dependencies other than method calls, such as import statements or field access. In one instance this led to HUSACCT detecting a violation that AJPOLog could not pick up on.

9 DISCUSSION

9.1 CLASS HIERARCHY AS ABSTRACTION

Throughout our approach we use the class and package structure as a hierarchy as an abstraction for objects. The interactions between these objects are method calls. While this abstraction works in demonstrating the rest of our approach, it is not without its problems.

First, it creates assumptions about the abstractions in the language(s) in which the system was written. We used Java throughout this thesis, and the ideas should translate well to C#. However, other programming languages have different ways of abstracting code units. For most programming languages, class hierarchy is not a usable abstraction and cannot be used.

Second and more importantly, the hierarchy of code units is generally not the best abstraction for software architecture. For code units to be a valuable abstraction, the programming languages the system is written in must all have some sort of architecturally significant level abstraction in them. For systems that involve different languages, these abstractions need to be somehow compatible for the architecture to be consistent.

Different views in software architecture require different visualizations, formalisms and abstractions. Salah and Mancoridis propose a hierarchy of dynamic software views, consisting of object interaction, class interaction, feature interaction and feature implementation, from bottom to top [18]. If we compare our work with their approach, we could consider AJPOLog as a tool for registering object interactions, and our ProM plugins to abstract those into class interactions. Because we have no way of knowing which interactions are related to which features, we cannot abstract any further along the hierarchy proposed in [18].

9.2 AJPOLOG SHORTCOMINGS AND ALTERNATIVES

For our case study, we developed our own instrumentation solution: AJPOLog. It meets the following requirements:

- Easy to use
- Widely applicable
- Logs calls, callers and callees

However, it was not without its issues. In the following sub-section, we will discuss the problems we ran into in our use cases.

9.2.1 Shortcomings

9.2.1.1 Instrumenting all classes

A drawback of AspectJ is that while it *can* weave arbitrary Java code (and even recompile JARs), it will not always do this without manual intervention. This means that if a Java project is configured to use external binaries, they will normally not be recompiled and instrumented. One external dependency is common to all Java programs and that is the Java standard library. Recompiling

the standard library manually is very difficult and error-prone [40]. Our own efforts to do so resulted in obscure error messages.

As a result of our difficulties instrumenting the Java standard library, we decided to accept that these classes would not be instrumented. Consequently, any call *to* a class in the standard library is registered, but any call done *within* the standard library is not. An unintended advantage of this is that logs do not become cluttered with large amounts of events that cover basic operations such as string concatenation and low-level data structure manipulation.

9.2.1.2 Unique object references

The objective of our method is to be able to model the interactions between individual objects. To identify individual objects, each object needs to have a unique identifier. The simplest identifier is the HashCode method built into the Java API. One downside of using this approach is that this interface guarantees that the same object will always have the same HashCode, but not that any two different objects have different HashCodes. Like the birthday paradox, the chance of two different objects sharing an identifier approach 50% at just 77,000 objects [41].

We do not think it is unrealistic for a fully featured Java application to have around 77,000 objects. Therefore, using the HashCode as a unique identifier would pose a risk to the accuracy of our models. Our initial solution was to generate a UUID for each object, which should greatly reduce the risk of duplicate identifiers. While this approach worked, there were several shortcomings.

First, adding an identifier to every object requires every object to be instrumented. As we have established, this is not possible for any object belonging to a class in uninstrumented external libraries. We therefore used HashCodes as a backup for objects for which adding the UUID field (silently) failed.

Second, adding a field to an object in AspectJ can only be done using inter-type declarations. Inter-type declarations can only have one target type, and in our experience, the members declared using inter-type declarations are not inherited by child objects. This means that the UUID field needs to be declared for every type that can be instrumented. Manually listing all the instrumentable types in a real-world system would be very time consuming, so we would need to find a way to do so automatically. Ultimately, due to time constraints, we decided that the combination of FQN + HashCode should be unique enough for our proof of concept.

9.2.1.3 Abstract methods

When we were using AJPOLog to instrument JabRef, we found that some log entries had ‘null’ as a callee. After further analysis we found that the call pointcut in AspectJ did not catch the fully qualified name of the actual object being called, but instead the signature that was being called. This meant that if the method being called was part of an abstract class, there was no reference to a concrete object. The FQN of that object was then ‘null’.

A quick workaround was to extract the name of the class from the fully qualified name of the method call. This is done by the conversion script.

9.2.1.4 *Log4j conflicts*

AJPOLog relies on Log4j for reliable and performant logging. However, this can lead to conflicts when the system under study uses Log4j for its own logging purposes. This is the case with JabRef, which resulted in log messages not created by AJPOLog ending up in the raw log file. Our log processing script was able to filter out these messages, negating the problem for JabRef 3.7.

JabRef 4.1 on the other hand configures Log4j in a different manner to JabRef 3.7. The configuration of JabRef 4.1 is created in such a way that it overrides the configuration of AJPOLog and nothing is written to a file. Changing the configuration mechanism of JabRef would require a significant change to its code, which is exactly what we wanted to avoid with AJPOLog.

An alternative would be to alter AJPOLog to not write to a file using Log4j, but instead write to a database. A database could provide additional benefits besides not conflicting with existing logging solutions. For example, it would allow for quick querying of events and greater reliability.

9.2.1.5 *Detecting all dependencies*

In our comparison of AJPOLog and HUSACCT in Section 8.6 we found that AJPOLog did not find an architecture violation because it only detected method calls. HUSACCT can detect additional types of dependencies, including indirect ones. We tried to implement logging field access in AJPOLog, but this led to an unacceptable drop in performance. We did not attempt to instrument other dependencies, but a dynamic approach like AJPOLog *should* be extensible to detect dependencies outside of method calls.

9.2.2 Alternatives

While AJPOLog suits our use cases, it is far from the only instrumentation approach available for Java programs. One of the earliest methods we found is BIT, which instruments Java bytecode [42]. The most recent and most advanced instrumentation method we found was DiSL[9] [43], which is a successor to MAJOR [40]. DiSL combines years of research on Java instrumentation for better performance than its AspectJ-based counterparts. It is also able to instrument classes in the Java standard library, which is normally problematic in AspectJ. However, despite our best efforts, it turned out to be very difficult to get DiSL working at all, and we did not succeed at getting it to instrument even a trivial Java program. Hopefully future releases will be more user friendly.

9.3 JabRef architecture violations

We used JabRef 3.7 for our experiment as this version was already used in academic research. The mapping of packages and classes to architectural elements we used was created in cooperation with one of the JabRef developers. While this allows us to create an accurate comparison of the implemented architecture and the intended architecture, this comparison is does not represent the current state of JabRef. Later versions are significantly refactored to better suit the intended architecture and should therefore have fewer violations. Furthermore, the mapping should be less involved than it is right now.

[9] https://disl.ow2.org/

We did try to instrument a newer version (4.1) of JabRef, but this failed. This is due to the aforementioned configuration issues when using Log4j2, where JabRef overwrote our configuration no matter what.

9.4 Concurrency and process mining algorithms

To make the concurrency in instrumented software explicit, we register the caller and callee of each method. However, to our knowledge, there are no process discovery algorithms in ProM that use this data to determine the precedence of events. For this reason, we used regular process mining algorithms that still use heuristics to recognize concurrency.

9.5 Shortcomings of ProM visualization

The visualization framework used in our Interaction Graph Visualizer uses ProMGraphVisualizer. This is a widget commonly used for visualizing diagrams in ProM plugins and is a wrapper for the JGraph visualization library. To represent the hierarchical aspects of the interaction graph we used ProMJGraph's support for 'containing' nodes: nodes that contain one or more child nodes and can be expanded or minimized at will.

9.5.1 Graph layout

While this worked well for very trivial graphs of about 4 nodes, the layout engine causes undesirable results even with our band example. Nodes are hidden behind other nodes, selection is difficult, and edges follow nonsensical paths. Considering that this example is trivial compared to a desktop application like JabRef, the layout engine is clearly insufficient for architecture reconstruction.

One workaround we employed while creating diagrams is to make nodes translucent. Originally, nodes had an alpha level of 0.9 – 90% solid and 10% translucent. When all nodes are expanded, it is not unusual to have more than six nodes on top of each other, each node only letting through 10% of the colors of the underlying node. Nodes that were drawn behind other nodes were therefore easily obscured. We changed the alpha level to 0.1, making previously obscured nodes more easily visible while still retaining some sense of depth in the nodes.

9.5.2 Collapsible nodes

Not only the layout of the graph causes problems when using the JGraph wrapper. Additionally, the API seems to not be designed for collapsible nodes. For example, it would be useful for us to receive events when a node is collapsed so that we can adjust the model underlying the graph that is being visualized. This does not seem to be possible.

Collapsed nodes also have undesirable effects on the way in which edges that are connected to them are drawn. When a parent node is collapsed, every child node's edges are drawn separately, as if the child nodes were still there. The only difference is that the edges seem to spawn from seemingly random places. Such behavior is detrimental for the user experience, as there are now multiple edges in the same direction between the same visible nodes, with no indication what differentiates the edges.

The problems described in the last two paragraphs make it difficult or even impossible to implement interaction graphs in exactly the way we set out to. The idea that edges can be fused whenever nodes are collapsed does not translate to our graph implementation.

9.5.3 Performance considerations

Aside from the problems with the generated layout, there are also problems with the performance of our approach. Converting logs form XES to Interaction logs, visualizing them as trees and selecting a sub-log goes quite quickly, taking a few seconds at worst for our JabRef case. Generating the graph visualization can take quite a bit longer. For the test log it only took a few seconds, but in our JabRef case we had to terminate the program after several hours.

9.6 TESTING INSTRUMENTATION METHOD

In Section 7.3 we check if our log matches our expected call graph. Nevertheless, several discrepancies between the two originate from our ignorance of the Java language. The question remains if this method of evaluation is useful if someone who has better knowledge of Java creates the call graph. We think this approach does not scale to larger systems, not only because the call graph (and log) would be much larger, but also because the true workings of the code would even further deviate from the expected workings.

10 CONCLUSION

For this thesis, our objective was to provide insight into the actual concurrent behavior of software within a software architecture derived from software execution data. We divided our research into a main question and four sub-questions. To conclude, we summarize the answers to our sub-questions:

> SQ1: What is the current state of the art in dynamic architecture reconstruction?

Several types of approaches can be considered state of the art in dynamic architecture reconstruction. We considered a set of approaches that had similar objectives to ours and found that none met all our own objectives.

Dynamic SAR approaches can be subdivided by their modeling techniques. On one side there are approaches that use formal models, such as finite state machines or Petri nets. These often have limited support for concurrency, either because the formalism doesn't support it or because their source data cannot explicitly represent it. Other approaches are less formal, creating functional architecture models, component and connector views or simple box diagrams.

> SQ2: What data needs to be collected to discover concurrent behavior from running software and how?

For our envisaged approach we need to collect method calls from running software. For every method call we register the caller, the callee and the method being called. Registering both the caller and the callee should enable us to register concurrency explicitly.

The way we register this data is using our own newly developed tool, AJPOLog. AJPOLog allows users to non-intrusively instrument Java programs and output the results to a log file. This log file must then be processed using a script, after which it is ready to be imported into ProM, a process mining tool.

> SQ3: How can we visualize software execution data in software architecture?

For visualizing software execution data, we introduce a new type of model: The hierarchical interaction model. These models are structurally similar to functional architecture models but adapted for visualizing the class and object hierarchy of object-oriented software. We also present the required logic for selecting a subset of all interactions and abstracting object behavior by collapsing nodes.

To provide more insight into object behavior, we describe how the software execution data can be converted into logs for process mining. We abstract object behavior into class behavior while separating interactions into process instances. By doing this, process mining algorithms in ProM will recognize behavior from different objects within the same class as the same abstract process.

When combined, HIMs and process mining allow users to both see interactions between software elements in their static context, as well as drill down to their low-level behavior. We implemented our approach in a set of three ProM plugins: one for converting regular ProM logs into Interaction logs that can be used for our two other plugins, and two visualization plugins that allow users to select subsets of the interactions.

SQ4: Is the proposed approach feasible in real-life systems?

After creating tooling to support our approach, tested the tooling to see if it could be used to reconstruct software architectures. First, we presented a method that included all the necessary steps, from creating a scenario and instrumenting a system to mining processes. Evaluating our approach on a very simple example program, we found that most of the results were as expected. AJPOLog successfully logged method calls, and the logged behavior mostly matched the behavior we thought the program would exhibit.

Of the visualization plugins, the tree visualization (representing the class/object hierarchy as an interactive tree) worked without problems. The graph (HIM) visualization had some issues with user experience and layout, but otherwise did work. Either plugin can be used to select interactions for process mining.

After validating our approach on a very small piece of software, we applied it to a fully-featured desktop application: JabRef. We created a short scenario that covers some basic functionality and executed it in an AJPOLog-instrumented version of JabRef. There are some conflicts between AJPOLog and the logging system included in JabRef, but our conversion script filters them out of the log that is used in ProM.

Like with the first system we tried, the tree visualization worked perfectly. Mining process models from selected interactions likewise worked without issues too. However, the graph visualization plugin was unusable: after several hours of waiting for it to create a graph, we still did not have any result.

Just as we did with our first case study, we used the information in our JabRef log to check if the implemented architecture matches the intended architecture. Whereas we created the intended architecture ourselves for the first system, we use the intended architecture in the documentation for JabRef. To properly compare the objects in our logs to the elements in the intended architecture, we use a mapping of classes and packages to architectural elements provided by one of the JabRef developers and one of the developers of HUSACCT. We find that half of the rules on which elements can have dependencies with other elements are violated in the implemented architecture.

Aside from comparing the implemented architecture we found to the intended architecture, we also compare our architecture conformance analysis to that of Pruijt and Dietz, who used HUSACCT to statically check architecture conformance. The theoretical differences between a static approach such as HUSACCT and a dynamic one like AJPOLog apply in practice as well: in the static analysis, there were nine intended architecture violations. AJPOLog found five of these same violations. Of the remaining violations, three were not caught by our approach because the scenario we used for creating a log did not cover the code that violates the intended architecture. This difference is fundamental to the workings of architecture reconstruction approaches that use static inputs, such as source code, compared to dynamic inputs such as logs of method calls.

Not all violations caught by HUSACCT that AJPOLog missed were as fundamental in nature. The fourth violation was instead uncaught because AJPOLog only detects method calls and not import statements or field access. Additionally, AJPOLog detected a violation that HUSACCT did not

catch. We do not have an explanation as to why HUSACCT missed these calls, as they can be detected using static analysis.

> MQ: What is a software architecture reconstruction approach that visualizes the dynamic architecture of an operational concurrent system with minimal developer involvement?

An SAR approach that visualizes the dynamic architecture of an operational concurrent system uses software execution data as an input. This data is captured in logs using a non-intrusive tool, AJPOLog, thus limiting the amount of developer effort required. These logs consist of method calls with their callers and callees registered such that we can reconstruct the class/object hierarchy of the system under study.

We created a toolset that can process these logs and use them in ProM plugins we developed. One of these plugins uses the converted software execution data to visualize the system under study as a hierarchical interaction model. This model allows users to abstract parts of the system and select interactions between software elements for process mining.

Our approach thereby visualizes the dynamic architecture of an operational system, although this visualization is not without issues. The amount of developer involvement is minimal and the logs we create can be used for creating hierarchical interaction models, process models and doing architecture conformance checking. Our approach therefore meets its objectives and can be used for a variety of software architecture reconstruction tasks.

11 FUTURE WORK

11.1 ABSTRACTIONS AND ARCHITECTURAL MAPPINGS

Abstraction is an important aspect of software architecture, and our approach allows for only two types of abstraction: objects to their classes and classes to their packages. A feature-level abstraction such as in [18] is not possible with our approach, because our method does not map objects or their containers to features. We did experiment with doing so in a separate bit of tooling in Section 8.6, though this tooling is rather ad-hoc and requires a pre-determined mapping of packages to features.

Our approach to mapping is a simpler version of a system like the one presented in [17]. We developed an ad hoc tool to apply the mapping directly to traces; it might be useful to instead integrate this functionality in ProM. An alternative to creating a mapping of code to features is to mark traces: executing a scenario that specifically uses a specific feature and labeling it as belonging to that feature. This is the approach presented in [18]. It is perhaps also interesting to combine feature identification approaches such as those reviewed in [44] to semi-automatically map source code elements to features.

11.2 IMPROVED LOGGING

In Section 9.2 we mentioned several shortcomings of AJPOLog, and some alternatives. Future research could go into finding reliable instrumentation methods for Java and perhaps other programming languages.

A more specific research area could be unique identifiers in Java. In Section 4.2 we explained that we assumed that the FQN of a class combined with the return value of `(identity)HashCode`, a built-in method in each Java object, should be unique enough for our purposes. The `identityHashCode` method should always honor a contract where a single object should have the same `identityHashCode` throughout its lifetime. Regardless, it is not a requirement that two different objects should have different `identityHashCodes`, meaning there is a risk of collision. As we touched upon in 9.2.1.2, the risk of collision reaches 50% with around 77,000 objects. Because we combine the `identityHashCode` with the FQN of the object's class, this risk should be reduced: the same class needs thousands of instances for a collision to become likely.

Nevertheless, we have not done any research to back up our assumption that collisions between FQN and `identityHashCode` combinations are uncommon in any sizable system. It could be interesting to research whether object identity collisions are a significant problem, and if so, what could be a solution. An alternative would be to manually add IDs to every object by keeping a global counter and assigning an ID as suggested by [45].

11.3 VISUALIZATION

In Section 9.5, we list several properties of the current implementation of the HIM visualization that could be improved. For future work, we feel that an improved approach would be based on a

different graph library than the old version of JGraph used in ProM. The library we used was outdated, slow, difficult to use and couldn't give us the results we were looking for.

11.4 STANDARDIZED INTERACTION LOG FORMAT

There is a standard representation for software event logs in ProM[46]. Currently, our ProM plugin uses its own proprietary representation for interaction logs which is not interoperable with other plugins. Adding support for the XES Extension would create interoperability with existing plugins. In addition, standard XES features like XML serialization would be possible by switching formats.

11.5 INTERACTION-AWARE MINING ALGORITHMS

To our knowledge, none of the available process discovery algorithms can leverage software execution data to create better results. Potential future research could go into creating algorithms that use caller-callee relations as partial orders. Throughout our approach we use several workarounds to make process mining algorithms create results that are more relevant to software architecture, such as abstracting objects into classes, abstracting object interactions into event names (Section 5.5.1) and creating a custom log format that separates interactions based on an underlying static structure in the caller and callee fields. It might be interesting to create a holistic approach where software execution data is treated as a 'first class citizen' and such workarounds are not necessary.

12 REFERENCES

[1] N. Rozanski and E. Woods, *Software Systems Architecture - Working with Stakeholders Using Viewpoints and Perspectives*, 2nd ed. Upper Saddle River: Addison-Wesley, 2012.

[2] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 573–591, 2009.

[3] D. Garlan, "Software Architecture: A Roadmap," *Proc. Conf. Futur. Softw. Eng.*, pp. 91–101, 2000.

[4] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, 2009.

[5] M. Leemans and W. M. P. Van Der Aalst, "Process mining in software systems: Discovering real-life business transactions and process models from distributed systems," *2015 ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst. Model. 2015 - Proc.*, pp. 44–53, 2015.

[6] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Commun. ACM*, vol. 54, no. 4, pp. 142–151, 2011.

[7] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE Trans. Softw. Eng.*, vol. 32, no. 7, pp. 454–466, 2006.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," *Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014*, no. Section 6, pp. 468–479, 2014.

[9] J. M. E. M. Van Der Werf and E. Kaats, "Discovery of functional architectures from event logs," *CEUR Workshop Proc.*, vol. 1372, pp. 227–243, 2015.

[10] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, 2007.

[11] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Second Edi. Boston: Pearson Education, 2003.

[12] J. M. E. M. Van Der Werf, C. Van Schuppen, S. Brinkkemper, S. Jansen, P. Boon, and G. Van Der Plas, "Architectural intelligence: A framework and application to e-learning," *CEUR Workshop Proc.*, vol. 1859, pp. 95–102, 2017.

[13] H. Haas and A. Brown, "choreography - Web Services Glossary," *W3C Working Group Note*, 2004. [Online]. Available: https://www.w3.org/TR/ws-gloss/.

[14] G. Decker and M. Weske, "Interaction-centric modeling of process choreographies," *Inf. Syst.*, vol. 36, no. 2, pp. 292–312, 2011.

[15] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[16] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst, "The ProM Framework: A New Era in Process Mining Tool Support," in *Applications and Theory of Petri Nets 2005*, no. 3536, G. Ciardo and P. Darondeau, Eds. 2005, pp. 444–454.

[17] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard, "Efficient mapping of software system traces to architectural views," *2000 Conf. Cent. Adv. Stud. Collab. Res.*, p. 12, 2000.

[18] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: From object-interactions

to feature-interactions," *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 72–81, 2004.

[19] A. W. Bierman and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," no. June, pp. 592–597, 1972.

[20] N. Walkinshaw, R. Taylor, and J. Derrick, *Inferring extended finite state machine models from software executions*, vol. 21, no. 3. Empirical Software Engineering, 2016.

[21] L. Mariani, M. Pezze, and M. Santoro, "GK-Tail+ An Efficient Approach to Learn Software Models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 8, pp. 715–738, 2017.

[22] M. de Leoni and W. M. P. van der Aalst, "Data-aware process mining: discovering decisions in processes using alignments," *Proc. 28th Annu. ACM Symp. Appl. Comput. - SAC '13*, p. 1454, 2013.

[23] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Event London*, vol. pages, pp. 215–226, 1989.

[24] C. J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," *Acsc*, vol. 10, no. 1, pp. 56–66, 1988.

[25] M. Leemans, W. M. P. Van Der Aalst, and M. G. J. Van Den Brand, "Recursion Aware Modeling and Discovery For Hierarchical Software Event Log Analysis (Extended)," *eprint arXiv:1710.09323*, 2017.

[26] C. Liu, B. Van Dongen, N. Assy, and W. M. P. Van Der Aalst, "Component Behavior Discovery from Software Execution Data," *2016 IEEE Symp. Comput. Intell. Data Min.*, no. December, 2016.

[27] W. M. P. van der Aalst, A. Kalenkova, V. Rubin, and E. Verbeek, "Process Discovery Using Localized Events," *Appl. Theory Petri Nets Concurr.*, vol. 9115, pp. 287–308, 2015.

[28] H. Y. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: a system for discovering architectures from running systems," *Proceedings. 26th Int. Conf. Softw. Eng.*, no. May, pp. 470–479, 2004.

[29] C. W. Günther and W. M. P. van der Aalst, "Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics," vol. 4714, no. Bpm 2007, pp. 328–343, 2007.

[30] R. P. J. C. Bose and W. M. P. van der Aalst, "Context Aware Trace Clustering: Towards Improving Process Mining Results," *Proc. 2009 SIAM Int. Conf. Data Min.*, pp. 401–412, 2009.

[31] G. Greco, A. Guzzo, and L. Pontieri, "Mining Hierarchies of Models: From Abstract Views to Concrete Specifications," *Lect. Notes Comput. Sci.*, vol. 3649, pp. 32–47, 2005.

[32] G. Greco, A. Guzzo, and L. Pontieri, "Mining taxonomies of process models," *Data Knowl. Eng.*, vol. 67, no. 1, pp. 74–102, 2008.

[33] S. J. J. Leemans, D. Fahland, and W. M. P. Van Der Aalst, "Using life cycle information in process discovery," *Lect. Notes Bus. Inf. Process.*, vol. 256, no. i, pp. 204–217, 2016.

[34] "JabRef website." [Online]. Available: jabref.org.

[35] T. Olsson, M. Ericsson, and A. Wingkvist, "Towards Improved Initial Mapping in Semi Automatic Clustering," *Proc. 12th Eur. Conf. Softw. Archit. Companion Proc.*, pp. 51–58, 2018.

[36] S. Herold, "SAEroConRepo," *GitHub*, 2018. [Online]. Available: https://github.com/sebastianherold/SAEroConRepo. [Accessed: 06-Nov-2018].

[37] S. J. J. Leemans, D. Fahland, and W. M. P. Van Der Aalst, "Discovering block-structured process models from event logs - A constructive approach," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7927 LNCS, pp. 311–329, 2013.

[38] “3rd edition | SAEroCon - The 5th Workshop on Software Architecture Erosion and Architectural Consistency,” 2016. [Online]. Available: https://saerocon.wordpress.com/3rd-edition/. [Accessed: 06-Mar-2019].

[39] L. Pruijt, *Instruments to Evaluate and Improve IT Architecture Work*. 2015.

[40] A. Villazon, W. Binder, and P. Moret, “MAJOR : An Aspect Weaver with Full Coverage Support,” *Rev. Investig. Desarro.*, vol. 11, no. July, pp. 46–60, 2011.

[41] S. Marks, “How do I prove that Object.hashCode() can produce similar hash code for two different objects in Java?,” *Stack Overflow*, 2016. [Online]. Available: https://stackoverflow.com/revisions/40936940/2. [Accessed: 12-Feb-2019].

[42] H. B. Lee and B. G. Zorn, “BIT: A Tool for Instrumenting Java Bytecodes,” *Proc. USENIX Symp. Internet Technol. Syst.*, pp. 73–82, 1997.

[43] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, “DiSL,” *Proc. 11th Annu. Int. Conf. Asp. Softw. Dev. - AOSD ’12*, p. 239, 2012.

[44] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: A taxonomy and survey,” *J. Softw. Evol. Process*, vol. 25, no. 1, pp. 53–95, 2013.

[45] A. Orso and B. Kennedy, “Selective capture and replay of program executions,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1, 2005.

[46] M. Leemans and C. Liu, “XES Software Event Extension,” pp. 1–11, 2017.

13 LIST OF FIGURES

14 LIST OF TABLES

15 APPENDIX A: FULL-SIZE FIGURES

Start Time	End Time	Thread	Caller	Callee	Message
2018-11-27T15:28:36,588	2018-11-27T15:28:36,593	[main]	org.architecturemining.program.example.band.BandPractice.Static	java.util.ArrayList.CallerPsuedoId: 1337344609	public boolean java.util.ArrayList.add(java.lang.Object)
2018-11-27T15:28:36,593	2018-11-27T15:28:36,594	[main]	org.architecturemining.program.example.band.BandPractice.Static	java.util.ArrayList.CallerPsuedoId: 1337344609	public boolean java.util.ArrayList.add(java.lang.Object)
2018-11-27T15:28:36,594	2018-11-27T15:28:36,595	[main]	org.architecturemining.program.example.band.BandPractice.Static	java.util.ArrayList.CallerPsuedoId: 1337344609	public boolean java.util.ArrayList.add(java.lang.Object)
2018-11-27T15:28:36,612	2018-11-27T15:28:36,613	[main]	org.architecturemining.program.example.band.BandPractice.Static	java.util.HashMap.CallerPsuedoId: 1739876329	public java.lang.Object java.util.HashMap.put(java.lang.Object, java.lang.Object)

Table 13: First four lines of the CSV-converted example log

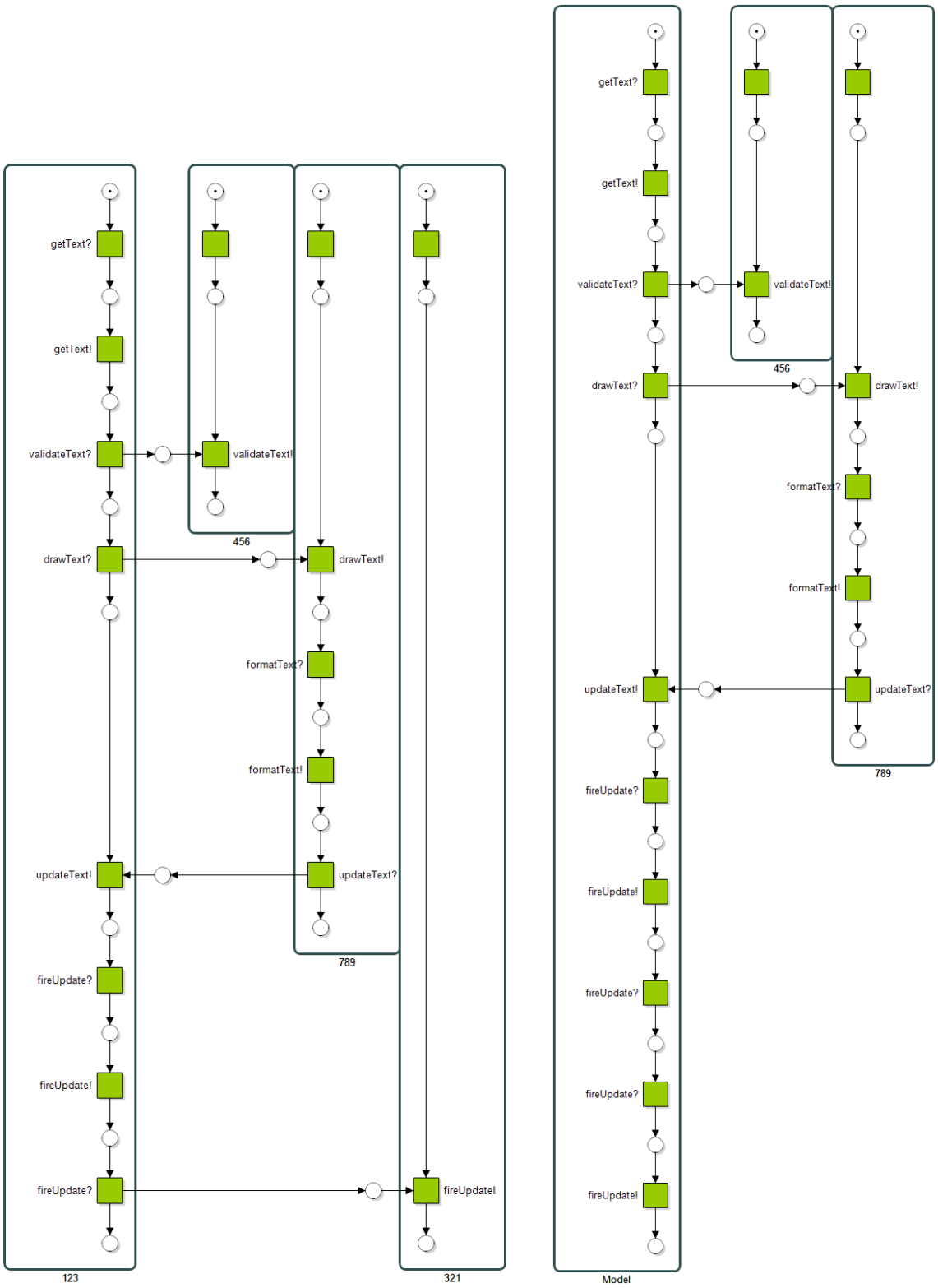


Figure 26: Comparison of the original running example (left) and the collapsed version

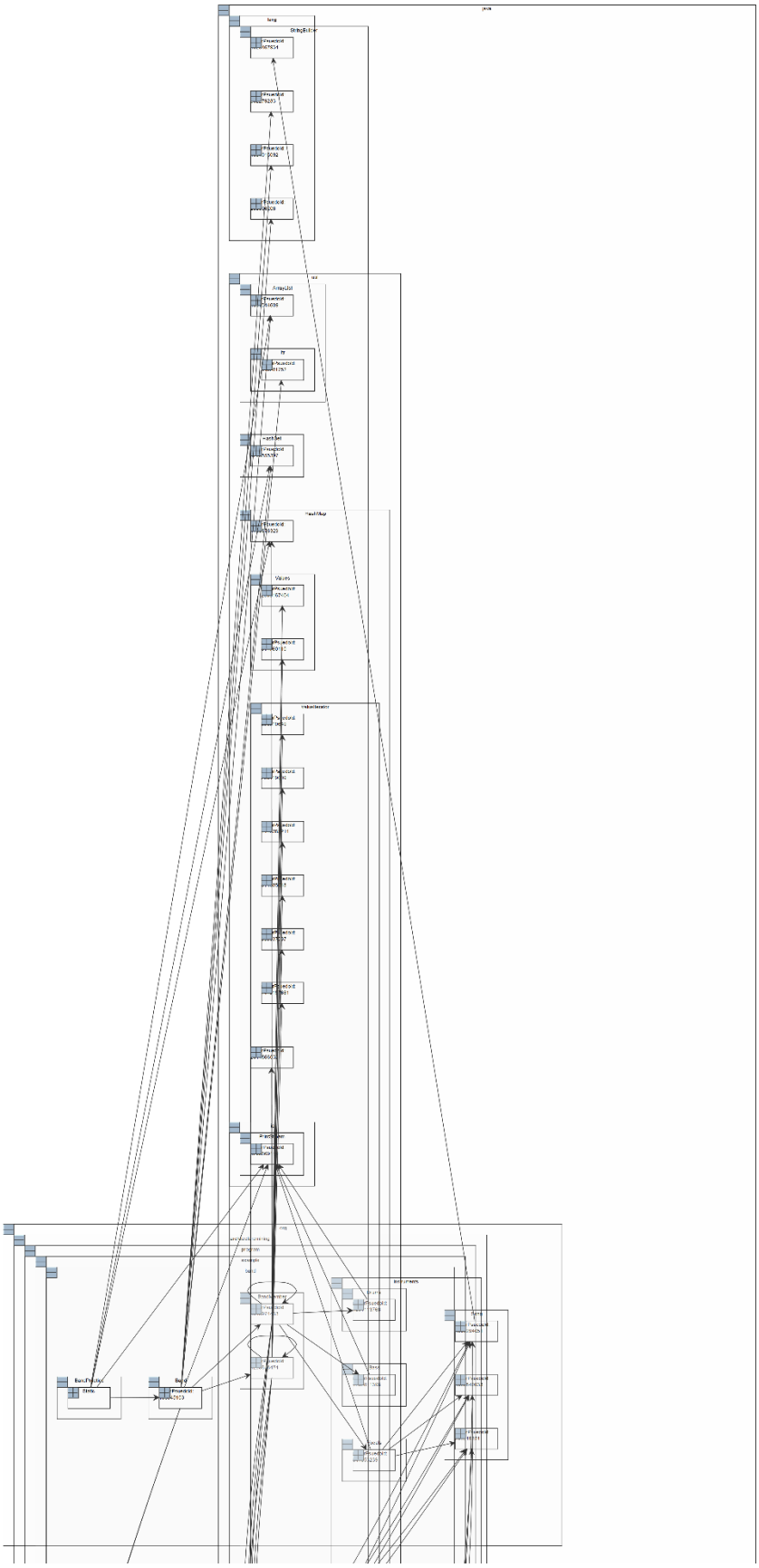


Figure 27: Full size ProM output of the band example

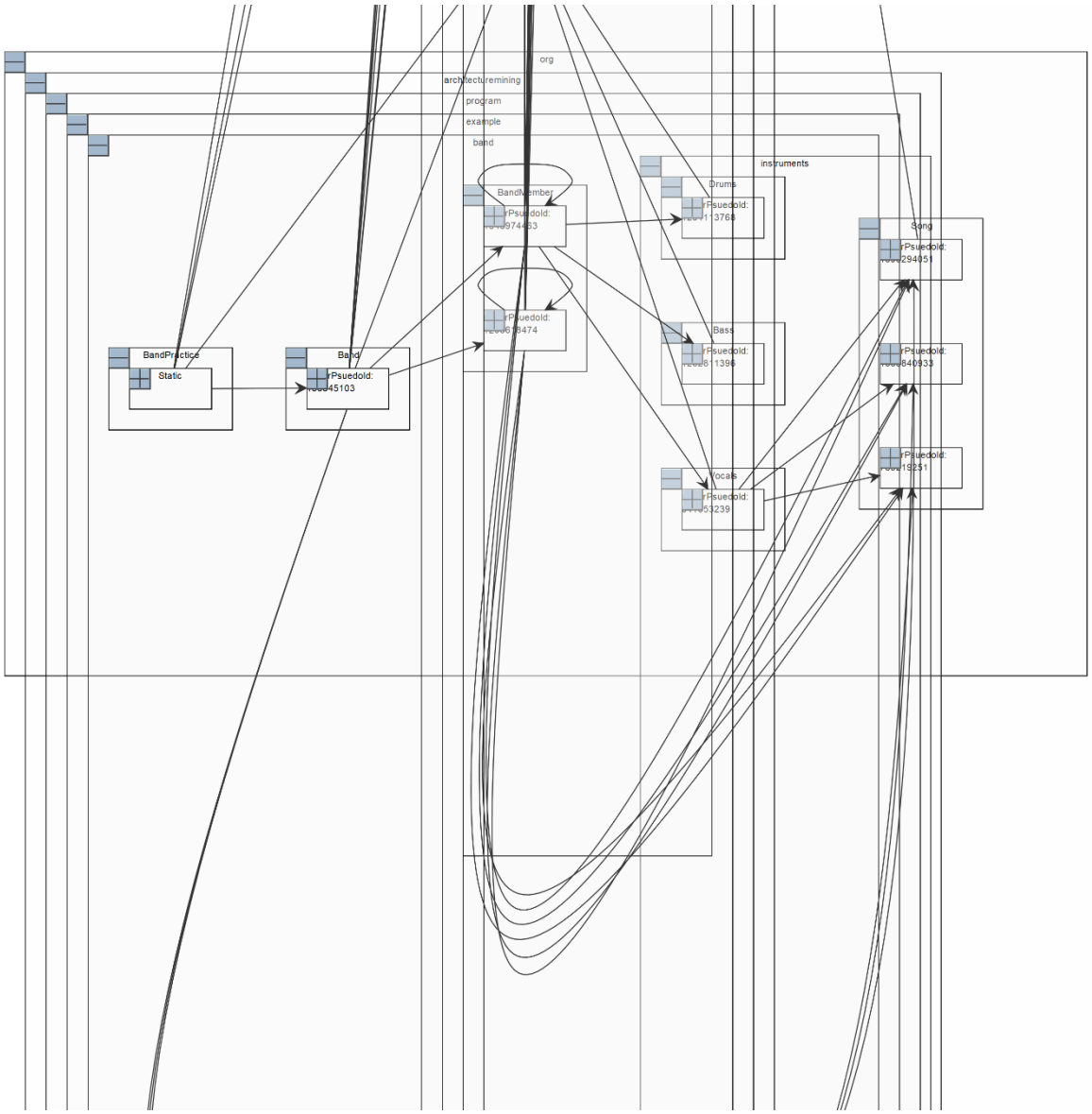
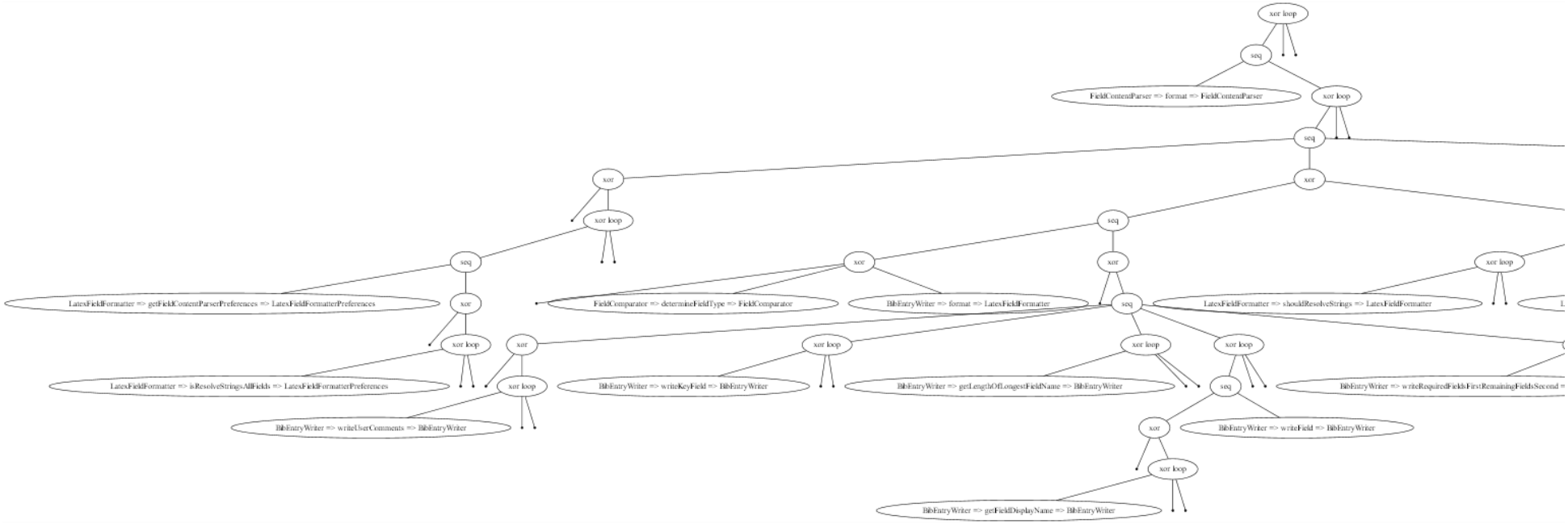


Figure 28: ProM output of the band example cropped to the 'org' package


xor loop
seq
FieldContentParser => format => FieldContentParser
xor loop
seq
xor
xor
xor loop
seq
seq
xor
xor
xor loop
LatexFieldFormatter => getFieldContentParserPreferences => LatexFieldFormatterPreferences
xor
FieldComparator => determineFieldType => FieldComparator
BibEntryWriter => format => LatexFieldFormatter
seq
LatexFieldFormatter => shouldResolveStrings => LatexFieldFormatter
xor loop
xor loop
xor
xor loop
xor loop
xor loop
xor loop
LatexFieldFormatter => isResolveStringsAllFields => LatexFieldFormatterPreferences
xor loop
BibEntryWriter => writeKeyField => BibEntryWriter
BibEntryWriter => getLengthOfLongestFieldName => BibEntryWriter
seq
BibEntryWriter => writeRequiredFieldsFirstRemainingFieldsSecond =
BibEntryWriter => writeUserComments => BibEntryWriter
xor
BibEntryWriter => writeField => BibEntryWriter
xor loop
BibEntryWriter => getFieldDisplayName => BibEntryWriter

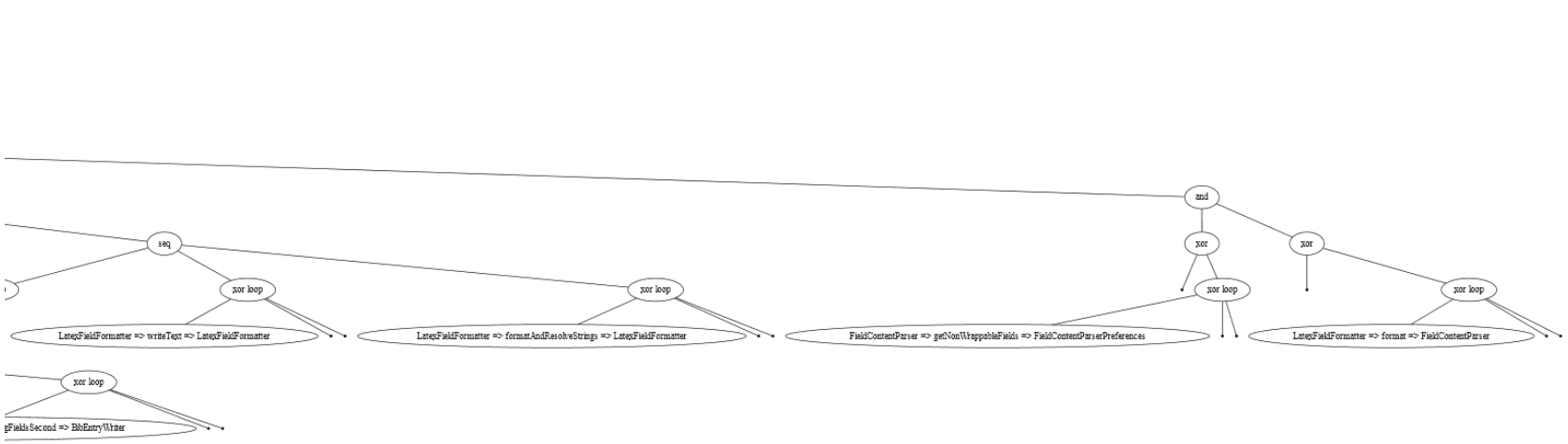


Figure 29: Full process tree of the net.sf.jabref.logic.bibtex package in our JabRef 3.7 scenario

net.sf.jabref.JabRefGUI	GUI
net.sf.jabref.JabRefMain	GUI
net.sf.jabref.collab.Change	GUI
net.sf.jabref.collab.ChangeDisplayDialog	GUI
net.sf.jabref.collab.ChangeScanner	GUI
net.sf.jabref.collab.EntryAddChange	GUI
net.sf.jabref.collab.EntryChange	GUI
net.sf.jabref.collab.EntryDeleteChange	GUI
net.sf.jabref.collab.FileUpdatePanel	GUI
net.sf.jabref.collab.GroupChange	GUI
net.sf.jabref.collab.InfoPane	GUI
net.sf.jabref.collab.MetaDataChange	GUI
net.sf.jabref.collab.PreambleChange	GUI
net.sf.jabref.collab.StringAddChange	GUI
net.sf.jabref.collab.StringChange	GUI
net.sf.jabref.collab.StringNameChange	GUI
net.sf.jabref.collab.StringRemoveChange	GUI
net.sf.jabref.gui	GUI
net.sf.jabref.migrations.FileLinksUpgradeWarning	GUI
net.sf.jabref.pdfimport.ImportDialog	GUI
net.sf.jabref.pdfimport.PdfFileFilter	GUI
net.sf.jabref.pdfimport.PdfImporter	GUI
net.sf.jabref.JabRefExecutorService	Logic
net.sf.jabref.collab.FileUpdateListener	Logic
net.sf.jabref.collab.FileUpdateMonitor	Logic
net.sf.jabref.logic	Logic
net.sf.jabref.shared.DBMSProcessor	Logic
net.sf.jabref.shared.DBMSSynchronizer	Logic
net.sf.jabref.shared.MySQLProcessor	Logic
net.sf.jabref.shared.OracleProcessor	Logic
net.sf.jabref.shared.PostgreSQLProcessor	Logic
net.sf.jabref.shared.event	Logic
net.sf.jabref.shared.exception	Logic
net.sf.jabref.shared.listener	Logic
net.sf.jabref.JabRefException	Model
net.sf.jabref.model	Model
net.sf.jabref.shared.DBMSConnection	Model
net.sf.jabref.shared.DBMSConnectionProperties	Model
net.sf.jabref.shared.DBMSType	Model
net.sf.jabref.shared.security.Password	Model
net.sf.jabref.Globals	Globals
net.sf.jabref.cli	CommandLineInterface
net.sf.jabref.migrations.PreferencesMigrations	Pref
net.sf.jabref.preferences	Pref

net.sf.jabref.shared.prefs	Pref
java.awt	Swing/AWT
javax.swing	Swing/AWT
java.sql	SQL
oracle	SQL

Table 14: Full mapping of JabRef 3.7 Java elements to architectural elements according to Tobias Dietz

16 APPENDIX B: HOW TO INSTRUMENT A PROGRAM

We will be using Eclipse, mostly to save ourselves from having to configure build tools and class paths manually.

16.1 CREATING A LOG

1. Find a Java program of which you have the source code. This will be referred to as the system under study (SUS)
    a. Preferably pure Java
    b. Preferably with clear instructions on how to build from source in Eclipse
2. Add the required instrumentation tools
    a. Install AspectJ tools for Eclipse
    b. Follow the build instructions for the SUS
    c. Make sure both AJPOLog and the SUS are correctly imported as projects in Eclipse
    d. Convert the SUS project into an AspectJ project (requires AspectJ plugin for Eclipse)
    e. In JabRef's case, for example:
        i. Follow instructions to setting up a local workspace (https://github.com/JabRef/jabref/wiki/Guidelines-for-setting-up-a-local-workspace)
        ii. Right click the project in the Package Explorer > Configure > Convert to AspectJ project
        iii. Right-click the project again > Properties > AspectJ Build > Aspect Path > Add Project > Select AJPOLog
        iv. Click 'Apply and close'
    f. By default, the log file will be saved in logs/scratch.log from the root folder of the SUS.
3. Create a scenario
    a. Define what parts of the SUS need to be included in the architecture model. In dynamic architecture reconstruction, only elements that appear in the log can be included in the generated architecture model.
    b. Create a scenario (a set of interactions between the user and the SUS) that covers these parts
    c. Run the instrumented version of the SUS
    d. Execute the scenario
    e. Stop the SUS
4. Extract the log
    a. Find the log file generated by the instrumented system
    b. Convert this log file into a more ProM-friendly format using a Python script
5. Convert the log into XES
    a. Load the modified log into ProM using the CSV to XES converter
        i. Make sure the fields are matched properly

b. Save the XES log as a .xes file, for easier future reference

16.2 CREATING A VISUALIZATION FROM A LOG

6. Import the XES file
    a. Make sure to use the naïve implementation of OpenXES (ProM log files – Naïve), as a bug in XESLite prevents the plugin from working properly
    b. Use the log as input for the plugin
    c. Select the correct fields for Caller and Callee
    d. Press 'finish'