Utrecht University
Department of Information and Computing Sciences

Architecture Mining with ArchitectureCity

Master Thesis

R.M. Rooimans B.Sc.

Supervisors:
dr.ir. J.M.E.M. van der Werf[†]
dr. J. Hage[†]
drs. F. van den Berg[‡]
dr. H. Vogt[‡]
[†]Utrecht University, [‡]Omnext B.V.

1.0

Utrecht, May 2017

Abstract

Ideally, software documentation follows the actual implementation. However, due to a plethora of reasons, many software systems have outdated or incomplete architecture documentation. In this paper, we present an approach that relies on the actual operation of software to gain new insights for software architects. Based on the software operation data generated by the system, we employ architecture mining to extract and enhance operational data to support the software architect. For this, we have developed the Architectural Intelligence Mining Framework, and more specifically, ArchitectureCity, which uses the analogy of cities to visualize the runtime of software: buildings, representing individual architectural elements are grouped in districts based on different clustering techniques, and streets depict the traffic between the different districts. We have applied the framework to a real life case study. The visualization techniques were positively received, which shows the potential of the proposed techniques.

Preface

I would like to thank Jan Martijn van der Werf immensely for the many enlightening meetings and messages sent over the past nine months, and for sculpting the research into what it is today. Secondly, I would like to thank Jurriaan Hage, without whom this thesis would have been entirely unreadable. I would also like to thank my external supervisor Frans van den Berg, as his help at Omnext was essential to the success of the research. Lastly, I would like to thank Laurens Duijvesteijn for proofreading the document, and for the many constructive discussions about the algorithmic solution within the framework.

Contents

List of Figures

Listings

Chapter 1

Introduction

Companies these days have Very Large Software Systems (VLSS) that have existed for a large number of years [39]. These legacy systems often support critical day to day business processes and cannot simply be replaced. Maintenance is becoming harder over the years as the technology used to build them is phased out, and programmers who still specialize in the technology become more sparse. Because of years of changes and additions, a legacy system is often poorly documented [29], especially in an architectural sense. Still, the risk of something breaking when replacing a system is deemed worse than the extra maintenance costs.

Maintenance is expensive and requires developers that are familiar with the old or proprietary software. Fixing or implementing features in a deprecated system is no long-term solution.

Knowledge about the system is one of the most important factors of success when maintaining or upgrading software; to be able to maintain something you need to know what it is. This knowledge can manifest itself in different ways [10, 24], e.g.;

- Architecture documentation
- Technical documentation
- Source code comments
- Working experience with the software

Software architecture is defined as "The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [8]. Reasoning about the system is exactly our goal, therefore, architectural documentation is extremely valuable.

The higher the quality and quantity of up-to-date knowledge available, the easier it is to maintain the software. The knowledge is often not equally up to date; architectural and technical documentation are, in many cases, far behind on the source code comments and experience levels. The latter are also the most volatile, as comments and source code change constantly, and experienced staff can leave at any moment. It is, therefore, crucial to have architectural and technical documentation to fall back upon.

As stated before, it would be helpful if developers know about all realized features in the code, or software units (SUs). In practice these SUs tend to drift apart from the intended architecture, thus leading to architectural erosion [24]. Keeping track of such erosion has few short-term benefits and is, therefore, often ignored. Identifying such artifacts at a later time is a difficult task indeed, but necessary for maintaining up-to-date documentation. Architecture compliance checking (ACC) tools can create a mapping of SUs to Architectural Elements (AEs), the implemented features mapped onto the architectural documentation. This the enables construction of dependency graphs to check for dependency violations [14]. When the AEs are mapped to the source code, this is called feature location. In the long run, this reduces technical debt and assists in the identification of code and increases consistency.

There are many downsides to having improper documentation: it results in higher maintenance costs, outdated security implementations and code inconsistencies. For example, it is incredibly difficult to extend a program that uses legacy code, or deprecated frameworks. When one encounters code written many years ago by programmers long gone, one often encounters deprecated coding standards or quick-and-dirty fixes. Additional time to locate the component and subcomponents that need to be updated or extended will have to be expected. It can be even more treacherous when old architectural documents do exist, as components can have changed since the documents were written. This greatly increases the chance of changing only a part of the problem, while not knowing about potential newer, undocumented parts. e.g. when a new adapter pattern is applied to a certain class and it is not documented, it can be very easy to miss and can result in extra code, code duplication, and inconsistencies in accessibility.

Not being able to use the latest releases of dependencies brings security risks. These days, many software updates contain security updates since most software is accessible through the Internet. When a software update will break the existing solution you have two choices: modify the solution to work with the software update, or not applying the update. Obviously, the latter only increases technical debt. Easy identification of components that should be updated can greatly reduce the modification effort required. Easy identification of open source components is also an important security aspect.

Feature creep is the demand for features in software that was never intended to contain those features. Most of the time, feature creep is a long-term problem, but it can also be the case that it manifests itself during the design process. Many older software systems are monolithic; all elements of the system are part of a single program. Feature creep often results in bloated software without proper architectural justifications. Monolithic systems are especially prone to this because they already contain a wide range of features. This makes the software difficult to maintain, poorly documented and hard to identify parts for re-use. The current trend is the creation of modular systems, or dividing the features of a monolithic application into modules. Dividing a system can be a difficult process, and therefore costly, but can improve performance, design, and implementation [9].

1.1 Objectives

In general, techniques like ACC are called software architecture reconstruction (SAR) techniques. We look at different SAR methods but mostly focus on process mining: the usage of event-log data for SAR purposes. The main advantage of process mining over conventional SAR techniques is that it captures dynamic aspects of a running system, while previous techniques only allow for static components. We are first and foremost interested in learning how we can capture dynamic aspects of a running system in order to construct relevant architectural documentation. This results in our research question:

RQ. How can we capture dynamic aspects of running systems to construct relevant architectural views and perspectives for stakeholders?

1.2 Running Example

As a running example in this thesis, we use a basic running application that logs the route and speed you ran, lets you save the routes to record progress, and displays an interactive map with a route overlay on top of it. It also contains an analytical component that calculates calories burned and average speeds on certain intervals.

The application is kept minimal on purpose, adding support for bike and snow sports would be an example of feature creep. You can argue that this is a nice feature, and in line with a sports monitoring app, but it is certainly not essential to a running application.

```
datetime,           package,                    processID,      actionID
15-09-2016 03:30:02,00 [runner.core.start]      (7)             Start
15-09-2016 04:36:05,00 [runner.core.run]        (7)             Run
15-09-2016 04:38:07,00 [runner.core.end]        (7)             End
15-10-2016 02:36:15,00 [runner.core.start]      (29)            Start
15-10-2016 02:42:05,00 [runner.core.run]        (29)            Run
15-10-2016 02:48:55,00 [runner.core.run]        (29)            Run
15-10-2016 03:22:25,00 [runner.core.run]        (29)            Run
15-10-2016 03:48:52,00 [runner.core.run]        (29)            Run
15-10-2016 04:30:01,00 [runner.core.run]        (29)            Run
15-10-2016 04:45:11,00 [runner.core.run]        (29)            Run
15-10-2016 04:57:05,00 [runner.core.run]        (29)            Run
15-10-2016 05:48:05,00 [runner.analysis.view]   (29)            View
15-10-2016 05:49:05,00 [runner.analysis.view]   (29)            Analyze
15-10-2016 05:54:05,00 [runner.analysis.view]   (29)            View
15-10-2016 05:54:45,00 [runner.core.end]        (29)            End
```

Listing 1.1: Part of the running app event log sorted by datetime, line 1-16. The entire log is found in Listing B.1 language

The application contains five points of logging, therefore, five actions that can be uniquely identified by process mining. These are *Start*, *Run*, *View*, *Analyze*, and *End*. *Start* is logged when the application is initialized, while *End* is logged on closing the application. *Run* is logged when the user starts a run, *View* as the user views a track they ran, and *Analyze* when the user analyzes the track they ran. Within the application, the system may contains many more actions, but for the purpose of process mining they are invisible, as they are not logged. Listing 1.1 contains an example log containing two uses of the running app. The log contains a datetime, the time of the event that was logged, a package id, the origin of the class that was responsible for the event, process id, an id of the system thread that handled the event and action id, the type of action that was executed within the event.

The possible state changes can be seen in Figure 1.1. The numbers in the arrows indicate the number of times the events transitioned in the order indicated by the arrow.

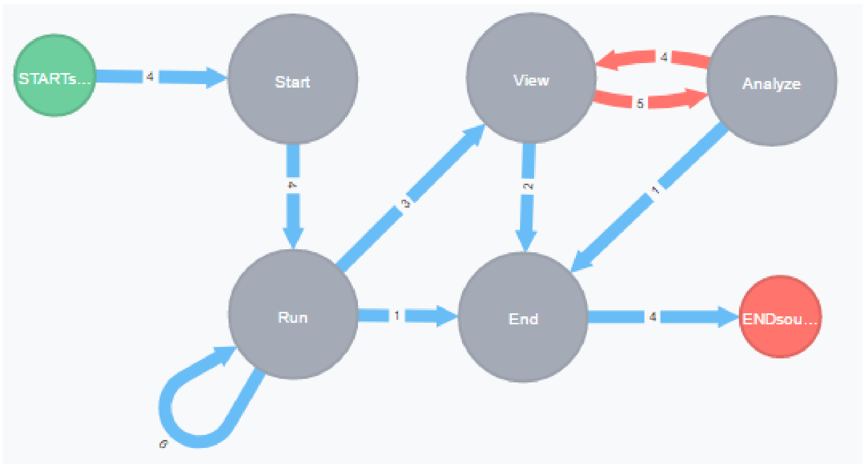


Figure 1.1: The flow visualization of the running example, manually rearranged for clarity. Blue arrows indicate a one-way sequence, while red arrows indicate a two way relation between the nodes.

1.3 Outline

We start off in Chapter 2 by stating the research questions. We then look at defining methods to document software architecture and give an overview of the state of the art of SAR in Chapter 3, discussing ways to use and produce reconstructions tools. We look at another usage of event-log data, the presentation of aggregated logs in a dashboard, which uses logs just like any other big data information stream. In Chapter 4, we present process mining, a small part of the complete SAR suite, but the main focus of our research. We give the specifications of our Architectural Intelligence Mining framework, AIM in Chapter 5. The AIM framework was built to conduct the process mining research in this thesis. Chapter 6 introduces the idea of ArchitectureCity, a variation on CodeCity based entirely on dynamic aspects of a system. In Chapter 7 we validate the framework and ArchitectureCity by applying the theoretical research to real world cases. Finally, in Chapter 8 we conclude the thesis and take an extensive look at future research.

Chapter 2

Research Approach

The main problem in software architecture reconstruction is the conversion of enormous files with many thousands, or even millions of lines of code, or lines of logging information, to a clear, and human understandable overview to gain new insights. When we limit the scope to architectural insights only, the domain is still enormous as can be seen in the idea we develop. Retrieving architecturally relevant information from dynamic data sources, such as logs, gives a new dimension to the traditional reconstruction from source code. These new insights will, however, require new methods of visualization to be useful to stakeholders.

We limit the scope to purely dynamic aspects of a system.

RQ. How can we capture dynamic aspects of running systems to construct relevant architectural views and perspectives for stakeholders?

As the relevancy of an architectural view depends on the concerns of stakeholders, we will need to define what we understand it to be. This immediately gives rise to the first sub-question:

SQ0. What are relevant architectural views to capture dynamic aspects of a system?

In order to make full use of the state of the art, we will look at the current landscape of software architecture reconstruction:

SQ1. What is the current state of the art in software architecture reconstruction with respect to dynamic aspects?

As we will see in the overview of techniques in section 3.2, the actual implementations of software architecture reconstruction can vary widely, as you can combine diverse goals with many different techniques. However, most techniques are used for static, not dynamic, information. This brings us to the following question:

SQ2. How to extract dynamic information from a running system?

Many visualizations exist which output relatively simple models based on log data alone. We will construct a dependency visualization that is easy to grasp for both management as well as programmers. It is scalable to display a high-level overview and more detailed low-level view for programmers to use when identifying dependencies or interactions. This leads us to the next sub-question:

SQ3. At what level of abstraction should we visualize the data?

This brings us to the log data itself; what do we see as dynamic information? What do we need from logs to qualify them as useful? Of course, the parsers created for AIM will need some basic data types to work, a process ID comes to mind to separate concurrent events within a single log file. This leads to the last question:

SQ4. Can we extract the level of abstraction at which we should visualize the data from the data itself?

It should be clear that the quality of the log directly corresponds to the quality of the architectural models that our framework, AIM, creates.

Research Methods

We begin with a literature study regarding the documentation of architectural information and the state of the art methods of obtaining such information. We look at the broad field of SAR literature and then focus on the process mining section and discuss the merits of its current applications.

For the validation of AIM, we employ an exploratory multi-case study of real-world software systems. These cases are concluded with an evaluation report formed from sessions with the stakeholders of the software systems for each case.

Chapter 3

Background

We start off answering SQ0 by defining and combining notation widely used in the architectural community. Section 3.2 looks at the SAR landscape and briefly touches upon the different techniques that exist by following the taxonomy of [14]. In section 3.3 we see that dynamic information comes in various forms, and can be used in various ways; aggregating log files gives insights on other levels than the SAR techniques but can be beneficial nonetheless. The last step in transferring information from logs to a human-readable format is the visualization. In section 3.4 we consider various visualization techniques currently in use along with their benefits and shortcomings.

3.1 Documenting Software Architecture

Software architecture is a broad term, therefore, we first need to define some notation. Each definition answers a part of SQ0, therefore, combining them answers the question.

SQ0. What are relevant architectural views to capture dynamic aspects of a system?

In Figure 3.1 we see the interactions between different artifacts related to a software system. As we want to capture dynamic aspects, we are mainly interested in the runtime artifacts.

We start with defining software architecture itself.

Definition 3.1.1. The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both [8].

In this section, we will go over some widely used software architecture structures.

3.1.1 Views

To talk about system architectures we need to introduce views.

Definition 3.1.2. A view is a representation of a set of system elements and relations associated with them. [11]

It is impractical to look at all systems elements, or SUs, at the same time, therefore, we only look at a small subset in a view. Such a view gives an architectural overview of a small subsystem within the system. A view can be seen as a conversion of SUs to Architectural Elements (AEs). Views are specific to a system, but can be categorized in two classes: static or dynamic. Static views cover static aspects of the system, such as relations between classes, and dependencies. Dynamic views cover, e.g., the response time of a query or the availability of a service. Collectively, the views describe the whole system.

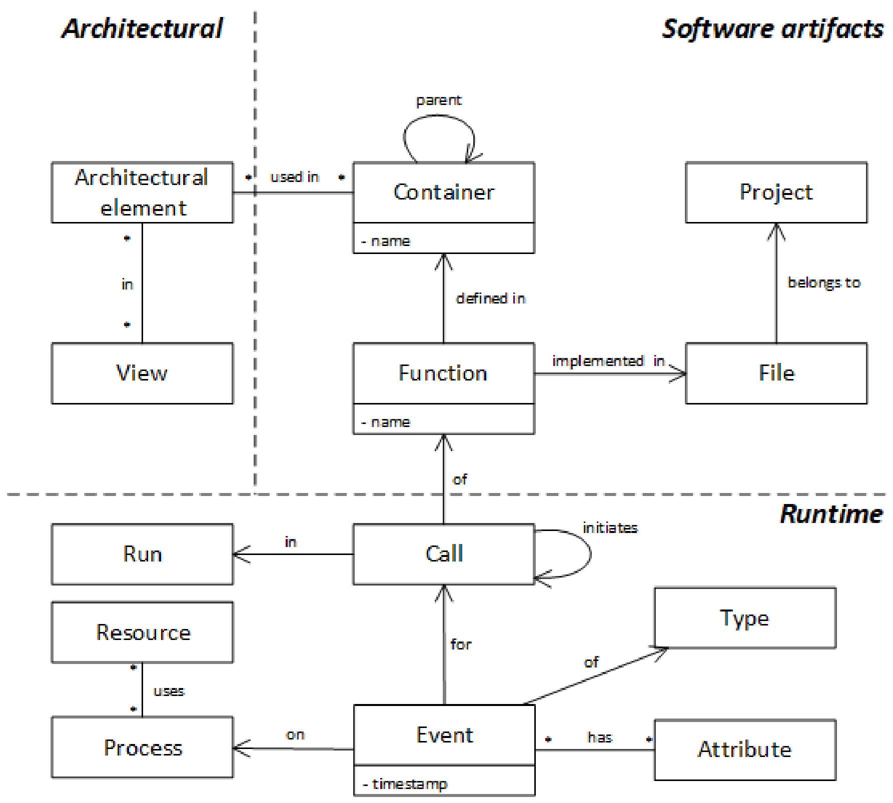


Figure 3.1: An overview of the interactions between different artifacts within the architecture, software artifacts and the runtime of a software system.

The two approaches for view construction are top-down and bottom-up; top-down works with a static set of views and tries to match it to the system, while bottom-up constructs views based on patterns found in the system. The views found should be mapped to viewpoints, that are in turn realizations of concerns of one or more stakeholders [11]. To provide minimal completeness there must exist at least one view for every identified architectural concern [22]. An example view of the running app can be the logic behind saving the route you ran to the database, described through different AEs.

3.1.2 Viewpoints

As views are tailored to a single system we need a term to talk about general categories of views: viewpoints [22].

Definition 3.1.3. A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views. [26]

Using viewpoints we can define categories of views to be reusable in many systems. The saving view can be part of a more generalized viewpoint for saving that uses general principles with concerns about e.g. consistency and formatting.

3.1.3 Relevancy

Whether a view is relevant is inherently subjective. Several questions arise, relevant for whom? When is something relevant? When the views are mapped to viewpoints and, therefore, to stakeholder concerns we can easily identify for whom a view should be relevant. The next question can be "what system elements would the view have to contain to be called relevant?". This depends on the concern and all conditions that apply to it.

Definition 3.1.4. Relevant views are sets of system elements and relations associated with them, that can be mapped to a concern of one or more stakeholders.

This is evident from the fact that all system elements should support a concern in some way. If this would not be the case these elements can be removed without a change in the realized concerns. Therefore, relevant *architectural* views are views that can be mapped to *architectural* concerns. We define relevance as having an impact on the stakeholder concerns. In the example of saving a running track, this would clearly be relevant as this relates to the stakeholder concern of retaining previously created data.

3.1.4 Dynamic Aspects

Dynamic aspects depend on software usage and are therefore not completely retrievable with static methods such as e.g. source code analysis. To obtain these aspects we need to use the dynamic output of the system, e.g., log files. Dynamic information visualizes the behavior of an example run of the program [21]. While the combination of static and dynamic aspects of systems provide the most interesting findings [21], we focus mainly on dynamic aspects and supplement static aspects from outside sources when it seems beneficial. An example of dynamic information regarding the running app would be the availability of the server that houses the stored track information. Static analysis will be of no use, as availability is not coded into the system but is mostly caused by software and hardware defects that are only visible in a running system [30].

Relevant architectural views to capture dynamic aspects of a system are sets of system elements and relations associated with them, that can be mapped to an architectural concern of one or more stakeholders and visualizes the behavior of an example run of the system. An example of such a view is a mail-server that has to send out mail within one second of it receiving the send-command. This is tied to the user stakeholder and satisfies their need for direct communication when requesting it.

3.2 Software Architecture Reconstruction

In this section, we will go over the possible methods to use within the field of Software Architecture Reconstruction (SAR) to answer SQ1.

SQ1. What is the current state of the art in software architecture reconstruction with respect to dynamic aspects?

The structure was taken from the work of Ducasse and Pollet [14], as this section will mostly cover the same scope. They divided the field along five different axes; goals, processes, inputs, techniques and outputs. In Figure 3.2 we see the different axes broken down further, each will be discussed in the sections below.

3.2.1 Goals

A goal dictates the type of information a SAR-tool will produce, and is, therefore, heavily connected to the intended use case. **Redocumentation** is the most common goal in SAR, as it produces new documentation of the software based on, e.g., source code or log files. When looking at source code or logs, you ensure that all the information retrieved is up to date [14]. **Conformance Checking** compares different sources of the same architectural documentation against each other to check for anomalies. It is common for software to experience architectural drift as it gets older,

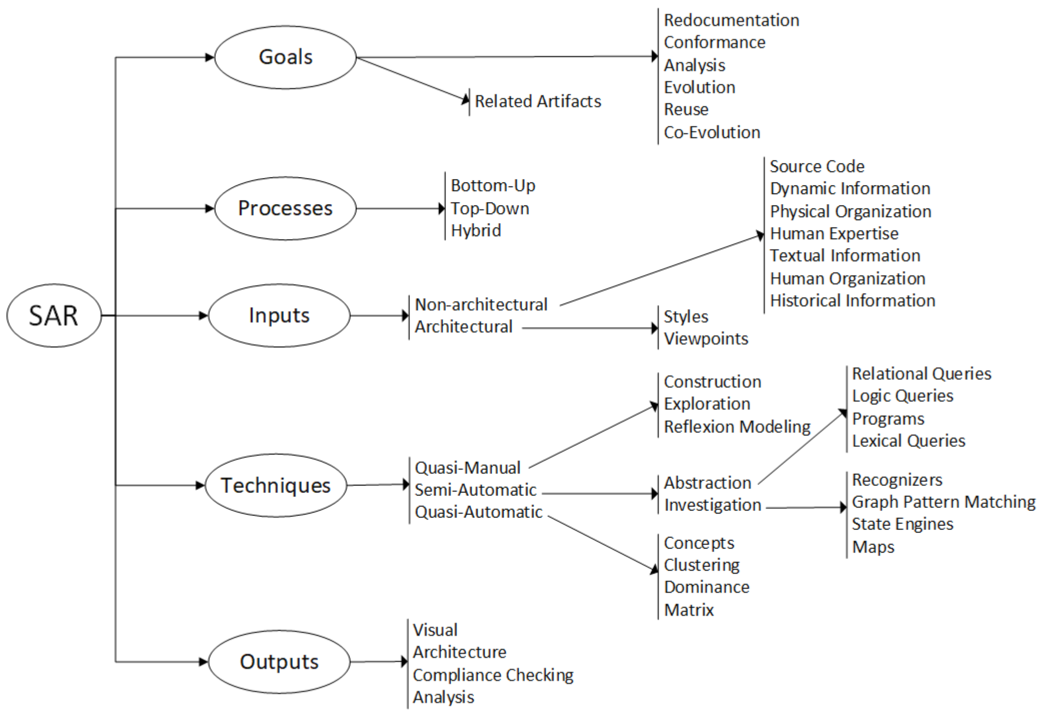


Figure 3.2: An overview of the SAR domain as proposed by [14] with reflexion modeling added.

but it is often hard to locate the issues. Conformance checking gives an overview of the violations, e.g., unimplemented but documented features, but more often implemented undocumented features. Conformance checking can also be done on dynamic aspects e.g. performance when using event log data as input [7]. **Analysis** can be e.g. dependency analysis or quality attribute analysis. This is incredibly useful when updating code, as a dependency analysis gives insight into the scope of the needed changes [14]. The goal can also be to support the **Evolution** of software; modern software often has an ongoing development cycle after release, as opposed to older software where updates were difficult and costly. A continuous feedback loop of system mining and actual implementation has been proposed [18], as to always possess up to date information. In the case of **Reuse**, we would like to identify commonalities between customized software products. By identifying where parts are identical, we can reuse these and thereby reduce costs, e.g. by converting them to services [14]. **Co-evolution** keeps track of the different speeds at which architecture and implementation change. They should be synchronized to avoid architectural drift [14].

3.2.2 Processes

There are three categories into which processes can fall; bottom-up, top-down or hybrid. When using a **Bottom-up** approach you start with the lowest level information and create a model. This step is repeated until a high-level overview is created. Bottom-up is used in most current SAR tools. **Top-down** starts with the higher level data and constructs hypotheses that are checked against source code. **Hybrid** approaches combine the two previous techniques. As both conceptual as concrete architectures are considered this method is frequently used to combat architectural erosion [14].

3.2.3 Inputs

Inputs can be split into two main types: architectural and non-architectural. To achieve any meaningful SAR you need non-architectural inputs, as reconstructing an architecture from existing architecture would simply be rewriting it.

Architectural Inputs

When available, an existing architecture can be invaluable in SAR as it can steer the process in the right direction. This includes validation or giving an initial architecture to expand. We recognize two different types of information; architectural **Styles** like data flow and pipes are widely used. They are general abstractions and therefore reusable in different situations. Next, we have **Viewpoints** as discussed in section 3.1.2. These architectural inputs are essential for architectural compliance checking.

Non-Architectural Inputs

The most obvious input is the **Source Code** itself, as it contains only true and up to date information. The code itself can be analyzed as text, but in most cased meta-data of the code is used [14]. **Textual Information** like source code comments, method or file names can also be included to extract information [14]. The core of process mining lies, however, in **Dynamic Information** such as event logs. It can be used to construct architectural views, just like source code, but also more in-depth analysis of performance, security, and availability. Depending on the information contained in the logs it can even show server load, the cause of it, and pinpoint the code responsible for handling the load [14]. **Physical Organization** of software often gives architectural information; source code files are almost always grouped in a tree structure that can be used to identify connected code. The **Human Organization** can also supply some information as according to Conway‘s Law “Any organization that designs a system will produce a design whose structure is a copy of the organization’s communication structure.”. Therefore, knowing the human organization can result in a better understanding of the system produced [12]. **Historical Information** is rarely used, and only used in (co-)evolution approaches [14]. **Human Expertise**, although not as trustworthy as other inputs, is still essential for SAR. Results need to be validated, and viewpoints selected. As many aspects of SAR are still subjective, human expertise is absolutely necessary [14].

3.2.4 Techniques

Techniques can be split into three categories based on their automation level; quasi-manual, semi-automatic and quasi-automatic. It is very well possible that a tool uses multiple kinds of techniques.

Quasi-Manual

With a quasi-manual technique, the tool merely assists the reverse engineer to understand their findings. A **Construction-based Technique** uses low-level information to assist the reverse engineer in manually abstracting the information [14]. **Exploration-based Techniques** focus on high-level features and guide the engineer through the realized code base [14].

Reflexion Modeling aims to identify the architectural drift by letting architects state explicitly their ideal architecture as edges and vertices to illustrate dependencies. The architect is then asked to map the source code to the previously made architecture [8]. The source code is then processed and any discrepancies are reported. Either the system or the architecture can be updated to more accurately match the system to the architectural documentation.

Semi-Automatic

Semi-Automatic techniques work with automated iterative refinement where the engineer merely steers the application in the right direction. **Abstraction-based Techniques** aim to map low-level concepts to high-level ones [14]. These techniques will be used in the framework proposed in Chapter 5 for the conformance checking. Engineers specify abstraction rules and the software iterates automatically. **Relational Queries** are used on relational databases. They use repeatable sets of transformations to construct architectural views [14]. **Logic Queries** extract relations and views with multiple-value queries [14]. Plain object-oriented **Programs** can also be used to perform certain steps in the SAR process [14]. It is possible to use **Lexical and Structural Queries** to make use of the inherent lexical and structural information contained in the source code [14].

Investigation based techniques map high-level concepts to low-level concepts, where high-level concepts range from architectural descriptions to design patterns. These techniques can be seen as highly advanced pattern matching algorithms [14]. **Recognizers** try to abstract these high-level concepts in the source code and returns all the places where such patterns are found [14]. **Graph Pattern Matching** does much of the same but transforms the patterns into graphs to make use of existing graph pattern matching algorithms [14]. **State Engines** track the system at run-time and report any architectural events that match predefined patterns [14]. **Maps** use rules to map hypothesized entities to source code entities [14].

Quasi-Automatic

Fully automated SAR tools do not yet exist, though quasi-automatic techniques come close. **Formal Concepts** aim to identify design patterns by utilizing lattice theory [14]. **Clustering** algorithms cluster objects or source code by their identifiers, like naming conventions and connectivity. This is a straightforward way to find tightly coupled components. Clustering also works well with dynamic information streams as the events can easily be clustered and analyzed [14]. **Dominance** uses graph theory; in a directed graph a node D dominates N if all paths from a given node to N go through D. This means that the component can only be used through another component if that component dominates it [14]. **Matrix** techniques expose dependencies and coupling by creating layers. A lower layer should not communicate with one above it. The lowest layers are, therefore, often helper classes or imported packages, as they are only called from above. The top level layer should be the application entry point, as all requests start from there; it, directly or indirectly, calls every other process [14].

3.2.5 Outputs

There are four types of output defined by [14]; visual, architectural, conformance and analysis. Of course, they all contain architectural components, but the scope, and therefore usage, differs.

The most widely used **Visual** outputs display the group's basic components and displays interactions between them, often in a simple diagram structure. Some focus exclusively on the source code as structure, and the only processing step is condensing the representation. Other tools extract more advanced structures and offer dynamic views with changing levels of abstraction or viewpoints. Some tools combine event logs with static code to create more viewpoints and offer more context [14].

Architectural outputs views, like a co-evolution view, often are more specific high-level views than the visual views. In the case of co-evolution, it can display a mapping of source code to intended architecture [14]. **Compliance Checking** uses two representations of the same system to check if they match, or what parts are different. This can be checking existing documentation against existing source code, but this can also be used to validate a newly mined architecture [14].

3.3 Dynamic Aspects of Reconstruction

The SAR techniques described above all require a certain input. Since we focus solely on the dynamic aspect of SAR we will look at the dynamic inputs and how to extract them.

SQ2. How to extract dynamic information from a running system?

```
datetime,               package,                    processID,      actionID
15-09-2016 03:30:02,00 [runner.core.start]          (7)             Start
15-09-2016 04:36:05,00 [runner.core.run]            (7)             Run
15-09-2016 04:38:07,00 [runner.core.end]            (7)             End
15-10-2016 02:36:15,00 [runner.core.start]          (29)            Start
15-10-2016 02:42:05,00 [runner.core.run]            (29)            Run
15-10-2016 02:48:55,00 [runner.core.run]            (29)            Run
15-10-2016 03:22:25,00 [runner.core.run]            (29)            Run
15-10-2016 03:48:52,00 [runner.core.run]            (29)            Run
15-10-2016 04:30:01,00 [runner.core.run]            (29)            Run
15-10-2016 04:45:11,00 [runner.core.run]            (29)            Run
15-10-2016 04:57:05,00 [runner.core.run]            (29)            Run
15-10-2016 05:48:05,00 [runner.analysis.view]       (29)            View
15-10-2016 05:49:05,00 [runner.analysis.view]       (29)            Analyze
15-10-2016 05:54:05,00 [runner.analysis.view]       (29)            View
15-10-2016 05:54:45,00 [runner.core.end]            (29)            End
```

Listing 3.1: Part of the running app event log sorted by datetime, line 1-16. The entire log is found in Listing B.1 language

3.3.1 Log Data

Software systems create enormous logs nowadays. Many systems create terabytes of data per day; a modern airplane creates terabytes of data per flight. Of course, the quality of the generated data is crucial; any system can output large quantities of low-quality data. The quality of the event log ultimately determines the quality of the process mining [40].

There are no set rules to determine log quality at this time, but there are two main directions one can take, aggregation or raw log output. When outputting raw log data one creates more data, but of lower quality, while aggregation results in more condensed logs. Which approach to take all depends on the ultimate goal for which the logs will be used. When applying process mining you would like many system events, but when displaying run-time statistics one can get away with less information. There are multiple ways to prepare the data, like pattern matching, normalizing and tagging [19]. Another way to pre-process the data is Artificial Ignorance where an AI removes all the normal behavior from the logs, leaving only abnormalities.

Log data comes in many forms, and not all are equally useful in process mining. Most logs, however, do share some key attributes. An example event log for the running app is shown in Listing 3.1, a small part of the entire log found in Listing B.1. The app has five different actions, namely Start (S), Run (R), View (V), Analyze (A), and End (E). The log contains a date-time, package path, process id, and an action id per entry. A process id is an id unique to the sequence of actions performed by a single user's interactions with the system under observation.

Anomalies within Logging Data

Not all events are necessarily logged chronologically, users will, e.g., sync their devices at different times. This problem is easily solved by sorting all events on their datetime attribute if it is available. When no datetime is present we can only assume that events are logged in order. Errors also occur, stopping a chain of actions early and, therefore, changing the end point of the transaction. Sometimes these errors are logged, and this behavior can be identified and solved by, e.g. ignoring the entire trace if it ends in an error. Other times these errors give extra insight and should be included in the final visualization.

Either complete entries or certain fields from an entry can be missing from the log data. The most straightforward way of dealing with such cases depends on the type of missing data. Most logs contain a certain amount of duplication, for example, every action id relates to the same package path. When a package path is missing, but the same action id always points to the same package we can use that information to repopulate the field. If a value like processing time is missing we can use statistical techniques to repopulate the field [27]. The worst case is a missing activity id or datetime. As inserting the wrong value of such a field can greatly affect the visualization we believe it is best to simply ignore such entries or even the whole trace. This again shows the importance of log quality for the analysis.

Sometimes essential properties are not logged, like the process id. The system thread and machine that are processing the action are often logged. These can be used instead of a process id in some instances, but this is heavily dependent on the system infrastructure. In the first case of chapter 7, we used the thread id instead of the missing process id, as the system could not easily be adapted to start logging a process id. In chapter 7.1, we discuss the limitations of this approach.

3.3.2 Big Data

In addition to the SAR method of using log files, it is also possible to take a more general, big data approach. Big data, as defined by [23], is large pools of data that can be captured, communicated, aggregated, stored, and analyzed. As data storage and processing power become cheaper, companies tend to store more and more data. It is projected that in 2020 there will be over 40.000 exabytes of information stored, about 5 petabytes per person. About two-thirds of all information nowadays is created or consumed by consumers, not large corporations. Only about one-third of the information in the world is deemed suitable for analysis in 2020 [15].

Information streams that are particularly interesting to analyze are surveillance footage, embedded and medical devices, entertainment, social media, and consumer images. Both the actual information and the meta-data is of enormous value [15]. Imagine a simple picture was taken: it contains GPS information, date and time as meta-data and future image analyzers can automatically identify every person in the picture, and even what the people are doing. When posted online this information can be enriched by user comments, expanding the context even further. A simple vacation photo of a sunburn may result in a targeted ad for after-sun, in the country that you currently reside in.

Non-Architectural Analysis

Many log analysis tools exist to process terabytes of log data per day. These tools, however, focus on a different type of log analysis. Products like Logentries[3], Loggly[4] and Scalyr[5] deal mostly with performance and error rates. They focus on time-based information to find information about e.g. throughput, error rates, and server load. These measurements, though usable in conformance checking, offer little architectural insight. While AIM uses mostly analysis, these types of systems use mostly aggregation techniques.

Most modern non-architectural analysis systems combine logs from the entire stack to form a single source for inspection. The human component remains relatively large compared to process mining, as it only summarizes and displays the data. These systems are a giant leap forward over conventional manual log analysis but are tailored to a different use case than SAR. An example of a non-architectural Loggly[4] dashboard can be found in Appendix A.1.

3.4 Visualizations

There are many possible ways to visualize architectural components ranging from simple matrices to dependency graphs and CodeCities. There is no single best visualization, as the goal of process mining can vary widely. Many techniques allow for multiple levels of abstraction, as they can be applied at, e.g., package, class or method level.

Petri nets

Petri nets are a formal, mathematical modeling language to represent business processes. A Petri net is a directed graph with two types of nodes: transitions and places. A transition changes the number of tokens in the connected places. A transition can only fire when there are enough tokens in the places that are connected to the incoming edges. An example can be seen in Figure 3.3; the places are the yellow circles and the transitions are the green and orange squares. An orange square indicates a XOR gate, meaning it only needs one of its inputs fulfilled, while the green square will require every input to contain a token, and will consume a token from every input place when fired. Actions that are ready to fire are indicated with a red outline, as can be seen in Figure 3.3. The transitions with an E and C are special; the E transition is an emitter that starts the process while the C is a collector that end it. For a process to be sound it should reach its termination state with the same number of tokens in its collector as there initially were in the emitter. Many Petri net discovery algorithms exist and are used for process mining. [13]

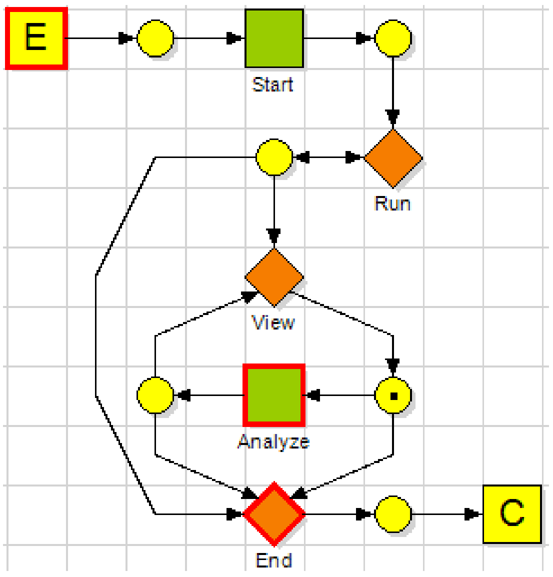


Figure 3.3: A Petri net made with Yasper[17] of the running app log file B.1 in the middle of a run.

The Petri nets language has been extended to the Extended Petri net language [25], adding, even more, features, like transitions that use every token from a place, or transitions that cannot fire if there is a token in a certain connected place. This extended version still allows for automated soundness checks.

CodeCity

The concept of a city-like representation of code, CodeCity, uses classes or methods as buildings to visualize a software system [37]. A city is a familiar sight to people and can, therefore, be efficiently be processed, even in complex cases. The classic CodeCity uses the package structure of the software system to structure the city. A single building often corresponds to a single class, although it is possible to create a building per method or function point. Using this style fits quite well with the object-oriented programming paradigm [38], as it requires only a simple one to one mapping. The buildings vary in footprint, height, and color to reflect internal properties. This makes CodeCity a versatile visualization technique, as it allows multiple dimensions to be clearly displayed. Another benefit is that CodeCity is language independent.

CodeCity allows for an interactive, 3D inspection of a software system. CodeCity tools often allow free movement throughout the city and offer additional information through tool-tips. Some

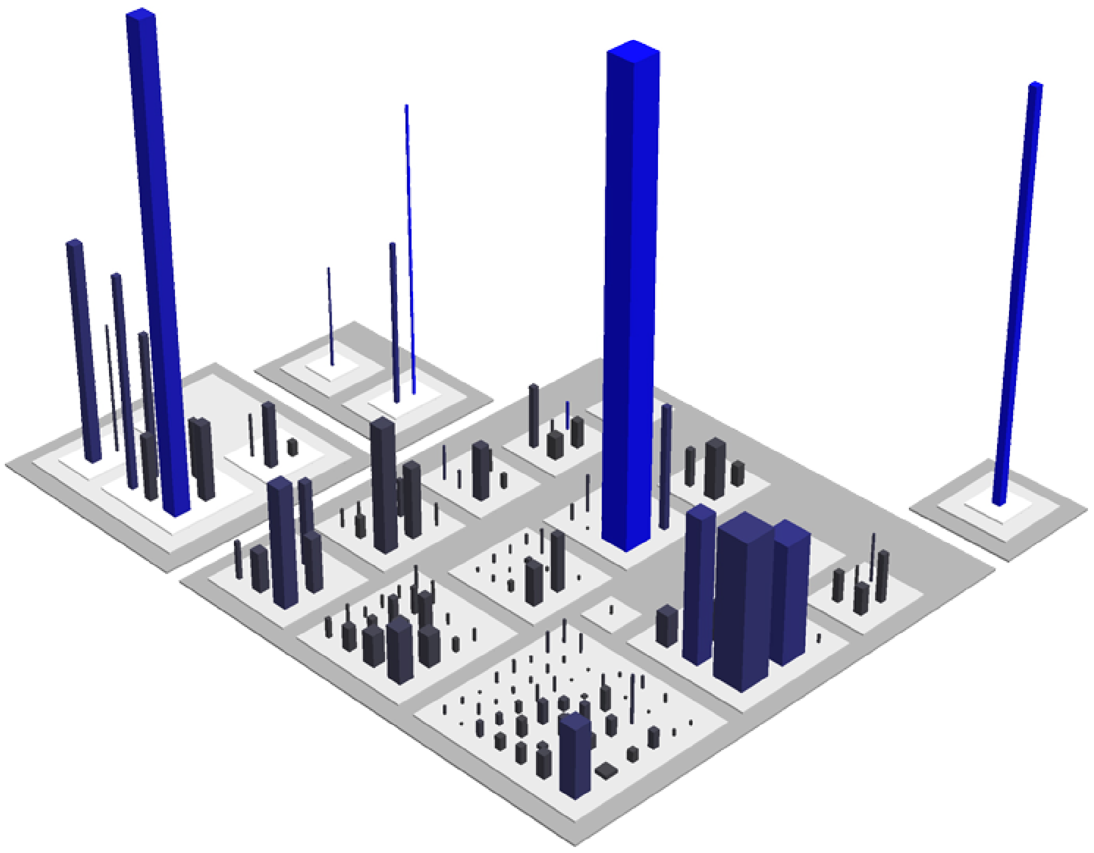


Figure 3.4: CodeCity representation of the CodeCity, v1.303 source code [1]

even allow filtering and other dynamic modifications like changing what code properties are tied to the visual properties of buildings. A CodeCity representation of the CodeCity source code can be seen in Figure 3.4. Even without knowing anything about the code we can clearly see a structure emerge, and we can pinpoint some key classes: the high blue stacks. Combined with deep knowledge about the system, the overview will become many times more valuable as more context is available.

Dependency graph

A dependency graph visually shows all dependencies on a given level, most of the time this is at class level. This allows for a clear overview of the entire program while still providing in-depth information. An example can be seen in Figure 3.5, where hierarchical edge bundling was used to create a clear and readable image. Most of the time the graph is directed, although this is not the case in Figure 3.5. The dependencies in a directional graph are transitive.

Dependency graphs tend to become chaotic when dealing with complex systems, as can be seen in Figure 3.6. Common techniques to reduce the chaotic nature of these graphs include edge bundling and clustering.

A variation on the dependency graph is the call graph, that creates a directed edge between two nodes when one calls the other. The visualization can be the same as a dependency graph in many cases, and it also suffers from the chaotic nature when systems become very large. Call graphs can be constructed from static assets like source code, or from dynamic aspects like logs. Comparing graphs from multiple sources is a graph matching architecture compliance checking problem. When all possible paths are run in an unchanged system, call graphs from multiple sources should be exactly the same. In practice, systems, dependencies and environments change

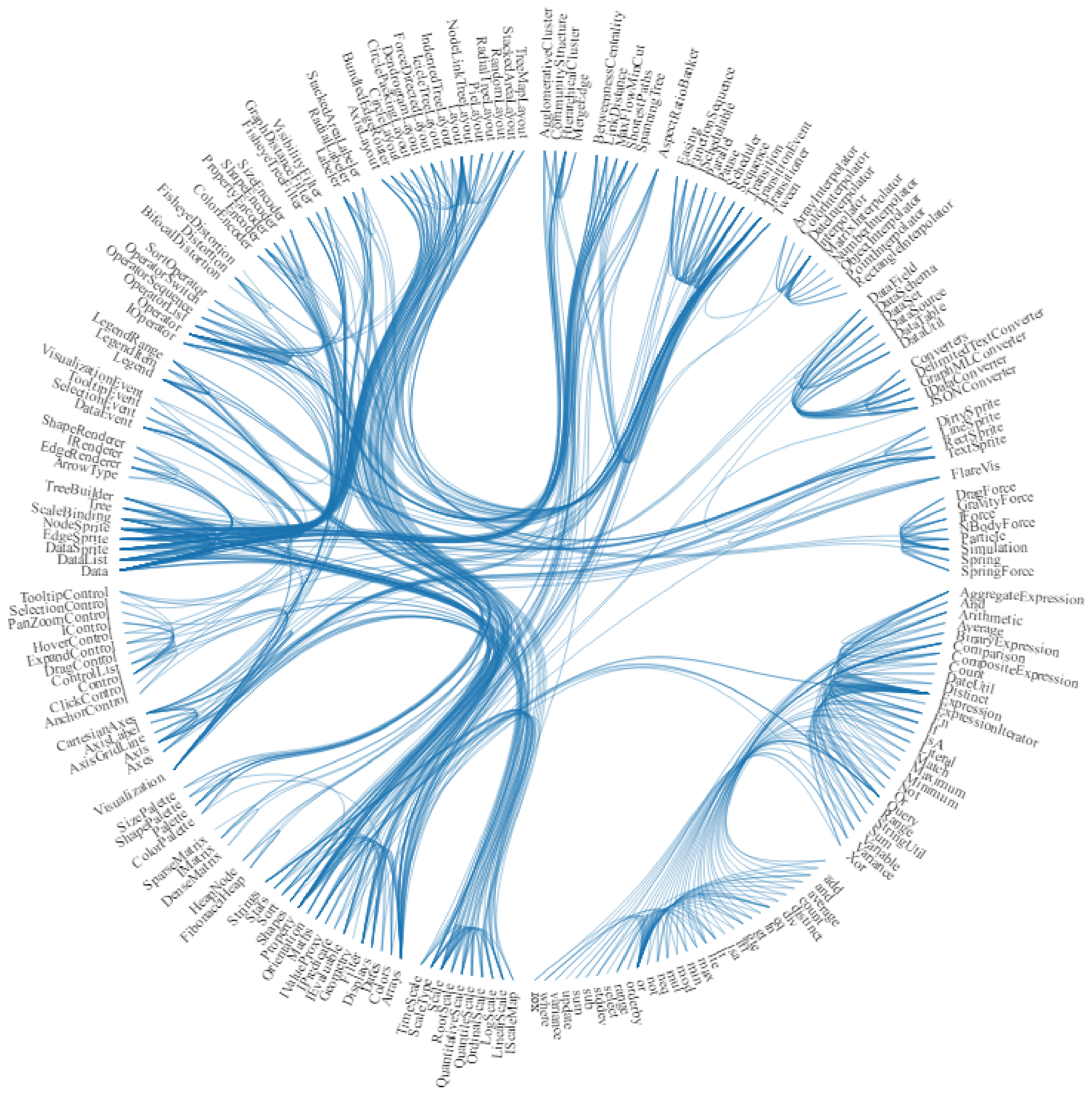


Figure 3.5: A hierarchical edge bundling dependency graph [2]

all the time, and it is very rare that all execution paths are run. Therefore, even properly generated graphs can differ.

Line width and color can represent a metric like as the tightness of the coupling or, in the case if dynamic aspects, call delays. Often, the nodes are color codes to indicate their origin or bundled together in the code package structure.

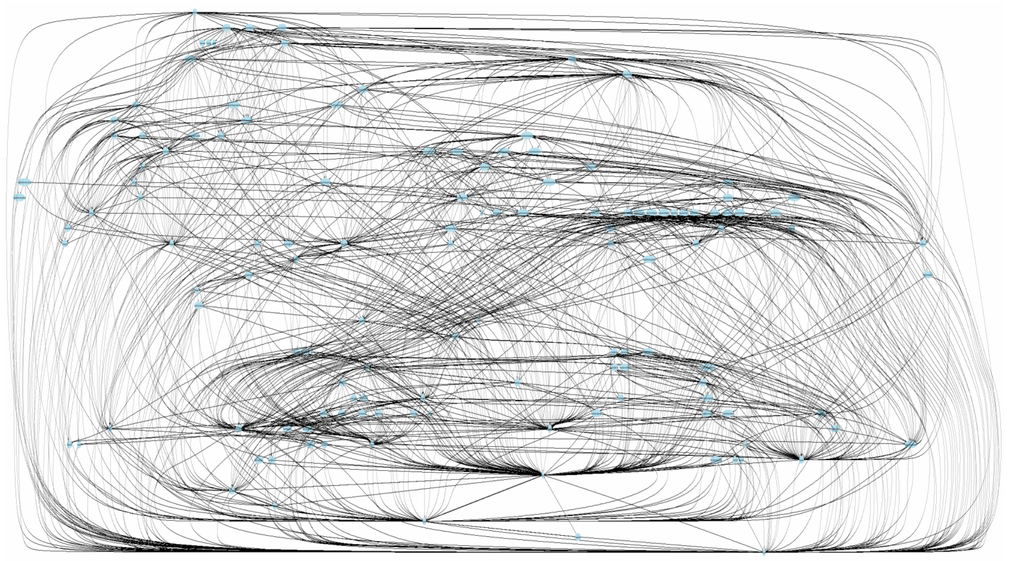


Figure 3.6: An example of a “spaghetti”-like dependency graph

Chapter 4

Extracting Architectural Information

The main focus of this chapter is to answer SQ2:

SQ2. How to extract dynamic information from a running system?

SAR provides us with methods to create a system architecture, but in most cases, some architectural documentation exist. As seen in Chapter 1 multiple levels of documentation exist, of which architectural documentation is just one.

A programmer with **experience with the software** is invaluable, but not always present. Employees may leave, and take crucial knowledge with them, and therefore you cannot rely on software specific experts to be your knowledge base. **Source code comments** give a low level, detailed explanation of what is happening. You will most often find these above method headers or next to complex calculations or calls. While great for understanding certain methods characteristics they will not give a high-level overview. This means that when you know where to look, this greatly assists you while maintaining or extending software. **Technical documentation** often contains an overview of the capabilities of the software product and is, therefore, a helpful place to start when looking for specific features. Information about the API ensures that extensions make use of the proper data streams, while knowledge of what functions exist reduces the chance of code duplication. **Design documentation** gives the highest level overview and is, therefore, a great place to start when working with a new piece of software. It should give a clear picture of the database structure and what classes contain what functionality. A listing of the dependencies is essential when charged with maintaining and upgrading software. Furthermore, the document may contain the reasoning behind some of the architectural choices.

In many cases, though, the software architecture is partly, or even completely, missing or outdated. Even new software might be missing an architectural design document because it might not have been needed when development started. While the software grew over time, it increased in size and complexity. For companies that have really organic growth, it might be smart to analyze and document the architecture and technical documentation before real problems arise. Reconstructing the architecture from scratch is a daunting task to do by hand; this is where SAR enters the picture.

4.1 Process Mining

Process mining takes dynamic system information, often event logs, and uses those to construct information about a running system. It aims to give insight into the processes, not necessarily how they are designed but how they actually behave. When compared to source code analysis, process mining has many advantages: it can capture dynamic aspects of the code and it is largely language independent. This means performance data like server load, wait times and error rates can be obtained and related to the source code. The downside of process mining over conventional static analysis techniques is that not all systems produce sufficient logging data, while most systems possess a complete code-base. In some cases, the system will need to be enhanced to support the level of logging that is needed for process mining.

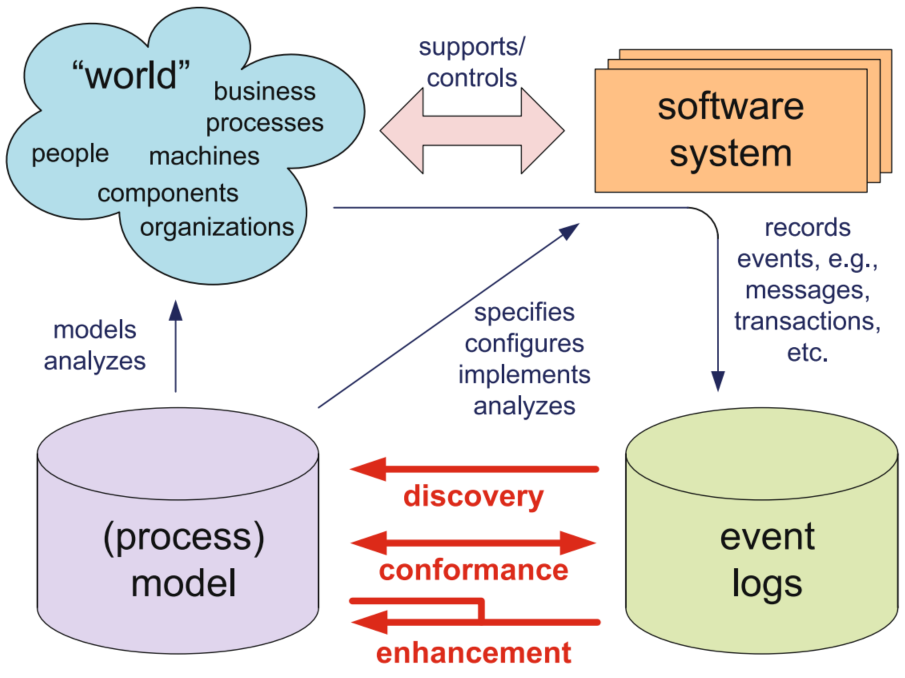


Figure 4.1: Positioning of the three main types of process mining: discovery, conformance, and enhancement[32]

.

Process mining can be categorized into three types: discovery, conformance, and enhancement. Discovery techniques take an event log and produce a model without any a-priori knowledge. All information the miner possesses is contained in the log itself. Every algorithms presented in Chapter 5 and Chapter 6 are examples of discovery based miners. Next, conformance techniques aim to identify misalignment between the existing process model and the model discovered from the event logs. Lastly, enhancement techniques utilize the event logs to improve the existing process models, this could be by repairing erroneous information, or by extending the model with newly acquired information [32].

Log Requirements

As process mining requires dynamic system information, the most straightforward solution is to use logging data. When logging is properly implemented the logs are flexible enough to be analyzed by generic software without too many modifications. It, of course, depends on the type and depth of the logs to what extent they are useful to analyze. If for instance, the logs do not contain precise ordering of events, the quality of the model produced will be much lower and it would be very limited in its capabilities.

We propose some minimally required information types;

- Event identifier: this can be implemented by giving every producing line of code an id, or can simply be the exact location of the event in the source code including file name, class, and method.

- Ordering: whether this is implemented using a logged datetime or just the guarantee that events happened in the same order as they are logged. This is essential to construct any flow diagram.

Datetime components also enable architectural performance analysis, as we are able to calculate the time between events. Clearly, if the logs contain more information the results will be more complex and in depth. If, for instance, we have call location information we can enrich the results with feature location. When we have access to e.g. server load statistics, we can display an additional layer containing the server load per action in the visualization.

Logging Standard

XES is an XML-based standard for event logs. The standard has been adopted by the IEEE Task Force on Process Mining as the default interchange format for event log [6, 16, 34].

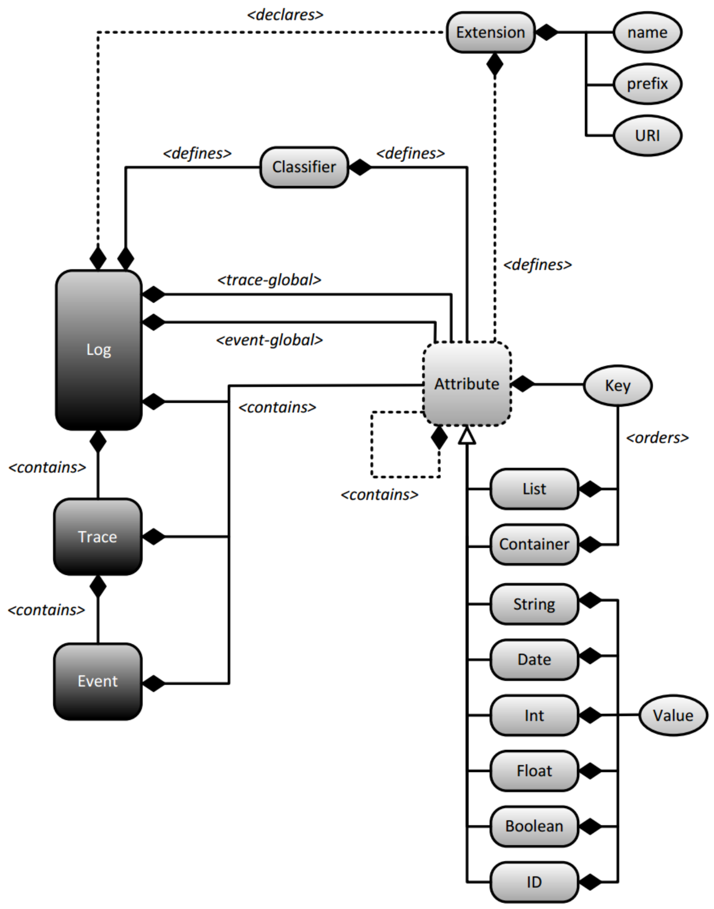


Figure 4.2: The UML 2.0 class diagram for the complete meta-model for the XES standard [16, 34].

.

XES uses a top level *Log* element that contains all information regarding a specific process. It also contains a number of *Trace* elements that contain information about a use of the process. Traces contain a number of *Event* objects that represent atomic granules of activity that have been observed during the execution of a process [16]. Logs, Traces and Events do not contain any information, they are structural elements. All information is contained in *Attributes* within the structural elements.

XES is used in the ProM process mining framework [33, 34] as a standardized logging format.

4.2 Monitoring Software Operation Knowledge

One of the must challenging tasks in software maintenance is to understand how software operates [28]. Software operation knowledge (SOK) has the potential to greatly assist in obtaining such understanding, but extraction and storage are often sub-par. SOK is often acquired utilizing unsophisticated methods, and implementations are ad hoc, application specific and exception triggered. This leaves valuable insights undocumented, while the data that was captured was often done using methods that are non-uniform between systems.

The three mayor issues with the usage of SOK are steep integration efforts, software specific implementations, and non-uniformity of resulting recordings. To tackle these issues Nuntia was proposed by van der Schuur et al. [28].

Nuntia injects logging statements into the byte-code of the system under inspection. This ensures that the tool is widely usable, as the underlying techniques are applicable to any language the allows binary instrumentation and reflection. What function points the acquisition logic is implemented varies on a case to case basis. When woven into the executed code, Nuntia starts logging method entries, method exits, and unhandled exceptions. Arguments for the method entries, return arguments for the method exits and stack traces and exception times are logged beside the id and datetime of the occurrence.

The final step is the visualization of the data gathered. Exceptions and methods are represented as nodes and their interaction as edges between nodes. Generic runtime information is also available e.g. total function calls or exception counts.

4.3 Architecture Mining

The principal task of a software architect is to design and develop a project strategy [31]. The software architecture is a combination of all the system views and their properties. Not every attribute of the views can be validated before the system is implemented, however, as some may rely on interactions between multiple components [35]. Methods like ATAM or simulation exist but remain approximations of true system performance.

This is why some researchers[35] advocate using software operations data, or dynamic aspects of the system, to close the loop between software architecture and realized software products; this is what they call architecture mining. Much like with SOK, a system is analyzed in a real world environment, and logging data is collected for further analysis, but now with a focus on architectural insights.

Definition 4.3.1. Architecture mining is the collection, analysis, and interpretation of software operation data to foster architecture evaluation and evolution.

This means architecture mining is the part of process mining that focuses on constructing architectural information and insights. The five main activities of architecture mining as proposed by [35] can be seen in Figure 4.3 as the blocks within the highlighted area;

- Architecture Reconstruction
- Evolution Analyzer
- Architecture Conformance
- Runtime Analyzer
- Architecture Improvement Recommender

The arrows indicate the flow of information between components, while the blocks are utilization or refinement steps. The bold, red arrows indicate which steps can benefit from using software operation data.

It is good practice to start software development with an intended architecture that is realized to create software artifacts, including source code and documentation. These artifacts are deployed into production and, once running, create software operation data. This operation data is often used to improve the realization and acts as a feedback loop. Low-level information like bug- and crash reports are often used to improve the realized software, but the system creates other types of software operation data as well. Architectural drift tends to occur when new features are realized that are not present in the intended architecture or there was no intended architecture to begin with.

Using log information containing execution traces, we reconstruct the architecture as the realized architecture with the techniques described in section 3.2. Note that this may be different from the intended architecture because of architectural erosion.

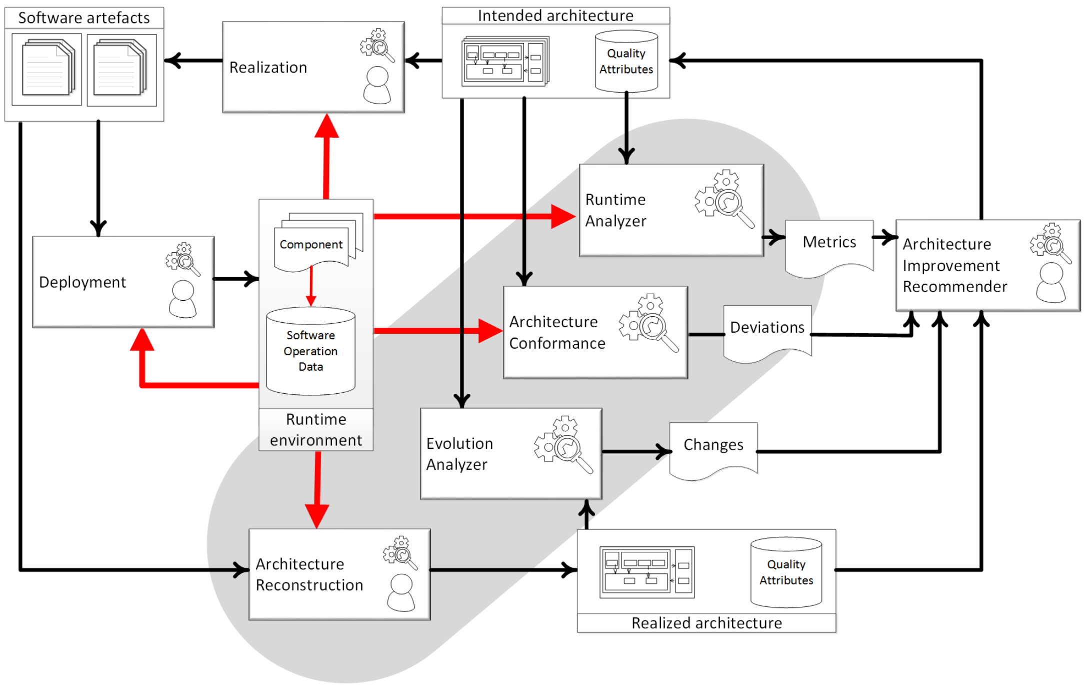


Figure 4.3: Conceptual model of the architecture mining framework proposed by Van der Werf et al. [35]

.

The evolution analyzer combines the intended architecture with the realized architecture to give an overview of the architectural erosion. Not following the intended architecture by the letter is not necessarily bad, but changes should be used to update the architecture documents.

The architectural conformance checker reports on deviations between the runtime environment and the intended architecture. This is mostly done by analyzing dependencies at source code level. It is unfeasible to overlay all dependencies of the realized software with the intended architecture, as this would result in the NP-hard subgraph isomorphism problem. By taking operation data into account during the design process of the system, the architect can limit the data that needs to be recorded, thereby allowing for better analysis of the system [35, 36].

Within the runtime analyzer, system properties are checked against quality attributes, such as performance and access control [36]. This type of analysis is often possible using aggregation techniques described in Section 3.3.2. The result of the analyzer is a set of metrics corresponding to the quality attributes defined in the intended architecture.

The metrics created in these components are combined to construct architectural recommendations which aim to improve the intended architecture. This closes the feedback loop. The authors of [35] envision a dashboard in which an architect is able to find continuously updated view overlays of the findings on top of the intended architecture which can be used to support the decision-making process based on the actual system usage.

Chapter 5

Tool Support for Architecture Mining

In this chapter, we propose Architectural Intelligence Mining framework AIM: a modular analytical tool that assists architects by giving them insight into the realized architecture of a software system. What sets AIM apart from most existing tools is that its only input is software operation data in the form of logging data.

AIM supports extensions for new types and standards of logging. Therefore, the system is split into three parts: **parsers**, **analyzers**, and **visualizers**. Parsers will transform raw log files to a generic predefined format: a combination of events, event-instances, and traces. All analyzers will use this format as their input. Therefore, any parser can be connected to any analyzer. The analyzers do not possess a common output format as the requirements for such a format are widely different depending on the analyzer.

The visualizers are, in nature, heavily tied to the analyzers; as of now, every analyzer has its dedicated visualizer. It is possible to create, for instance, a Petri net analyzer and display it using the existing Petri net visualizer, or the other way around, display a Petri net using another visualizer by coupling it to the existing analyzer.

The modular structure was based on the ProM framework for process mining, that also first parses logs to a common format, in their case XES, before further analysis [7, 34].

5.1 Technology Stack as Built

The AIM framework is written in C# and builds on the ASP.NET Core framework. It uses Semantic UI for CSS and Javascript in the web interface, backed by a Postgres and a Neo4j database. C# was chosen because it offers high performance, and in combination with the open source ASP.NET Core it offers a cross-platform framework with a fully customizable web interface.

The NuGet package manager is used to install and update packages. Besides the standard Microsoft packages, we use MoreLINQ for extra LINQ methods (while not absolutely necessary it allows for cleaner code), Neo4jClient as the .NET client binding for Neo4j, and Npgsql as the .NET client binding for the PostgreSQL database. These two bindings are essential for allowing database access to Neo4j and PostgreSQL.

We use QuickGraphPCL to allow for Quickgraph integration which contains the Graphviz open source graph visualization software. The DOT language is essential for the clustering visualizations we will see in section 5.4.

OpenSCAD is used in the visualization of ArchitectureCity and needs to be installed as a separate program to view the .scad files that the visualizer generates. There are no direct dependencies within the code base, and a switch to another 3D modeling tool can be made by replacing the small `OpenSCADExport` class with another exporter. OpenSCAD was chosen because it offers the programmatic creation of 3D objects with just the help of a 65 LOC .scad file, namely

`Building.scad`.

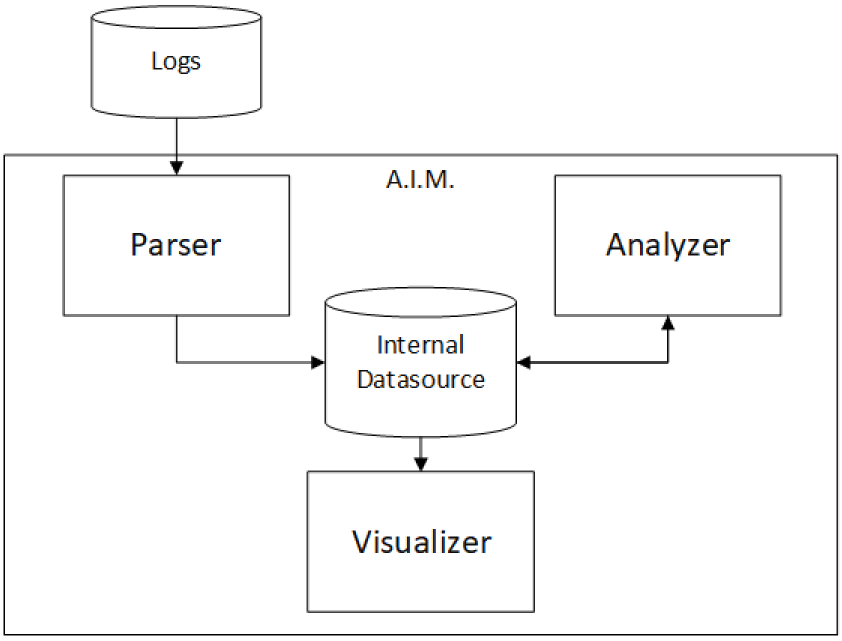


Figure 5.1: The data flow within the AIM framework.

5.2 Parser

As mentioned before, parsers have a common output format, but a widely different input format. A parser is the only class that needs to be written when dealing with a new type of logging, as it may contain domain-specific components. Parsers are generally also the smallest in terms of lines of code. The parsers constructed for the two cases we see in chapter 7 are 207 and 125 LOC respectively, and mostly contain domain specific information without complex algorithms.

5.2.1 Events and Traces

Every parser outputs two lists: one containing all the traces and one containing every unique event, that, together, contain all relevant information from the log file. This means that all Analyzers are initialized with these two lists.

The `Event` objects are log entries without any variables. They correspond to one single logging command in the source code but only contain information that remains the same over multiple log entries, like action id, and package structure.

Dynamic log information is contained in the `EventInstance` class; they are composed of an event, datetime and a thread id. One `EventInstance` is created for every log entry in the log, while one `Event` is created for every unique logging statement in the source code. This is different from XES as we have seen in Section 4.1. This structure resulted in over 80% reduction in memory usage over duplicating all static event information for every event instance, as events often correspond to many hundreds, if not thousands, of event-instances.

Traces are sequences of event instances, ordered chronologically by datetime. When two events have the exact same datetime, they are put in the same order as they were encountered in the log file. Ideally, sequences contain a single concatenation of events that form an action, or sequence of actions, performed by a user or automated process. Grouping on single processes is easily achieved when an activity id is logged, as we only need to group event instances by the activity id and sort them to create the traces.

5.2.2 Algorithms

Parsers are initialized with the file-path of the log that has to be parsed. The log then enters pre-processing: the file structure needs to be hard-coded into the parser, and values may need sanitation as they might contain variables. An example log is shown in Listing 5.1 were the actionID column contains multiple start actions combined with user names. These types of variables need to be removed in a pre-parse step, as each name-start combination would show up as a separate node in the visualization. We use regular expressions and if-else chains to identify and modify these variables and reduce them to their common parts. This ensures that the event-instances created for every `Start` action contain a reference to the same `Start` event.

```
datetime,            package,               processID,    actionID
15-09-2016 03:30:02, [runner.core.start]    (7)           David Start
15-09-2016 04:36:05, [runner.core.run]      (7)           Run
15-09-2016 04:38:07, [runner.core.end]      (7)           David End
15-10-2016 03:36:15, [runner.core.start]    (29)          David Start
15-10-2016 03:42:05, [runner.core.run]      (29)          Run
15-10-2016 03:48:55, [runner.core.run]      (29)          Run
15-10-2016 05:54:45, [runner.core.end]      (29)          David End
15-10-2016 14:00:00, [runner.core.start]    (17)          Peter Start
```

Listing 5.1: Variables contaminating the actionID column.

For every log entry, an event-instance is created and coupled to an event. Then, the `Traces` are constructed by grouping the event instances by process id. Lastly, every trace is sorted chronologically by datetime to make sure that even when events are logged out of order, the traces reflect their true ordering.

In some cases the process id is not available, so we group the instances on their thread id. This is not perfect, as multiple sequences running on the same system thread are seen as a single sequence. To mitigate this problem we further split the traces any time there is a preset time interval between two consecutive event instances on the same thread. The actual split-time between instances will depend on the actual system.

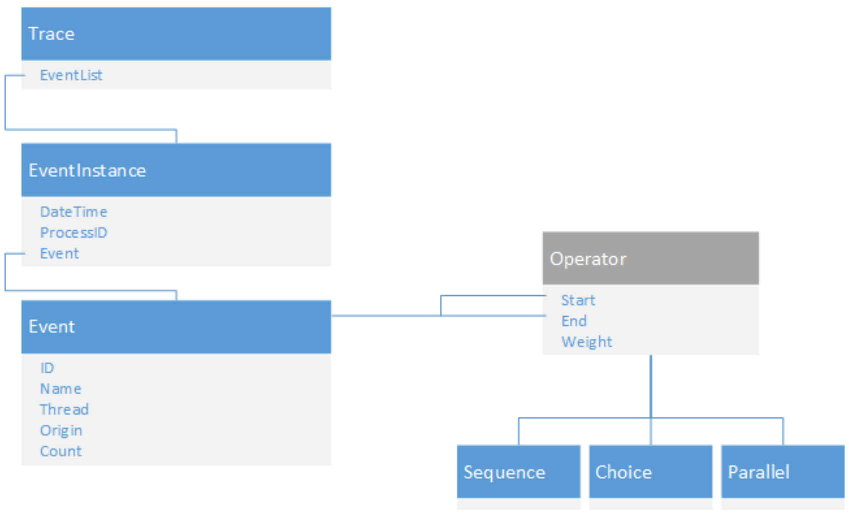


Figure 5.2: The class diagram of Traces, Events, EventInstances and Operators.

5.3 Analyzer

Analyzers will use the dictionary of traces and the eventlist generated by the parser to extract properties of the system. We mainly focus on application flow, but this is certainly not the only direction that analysis can go; if the log contains performance data one can write an analyzer to calculate e.g. function latency and message delays. Some analytical packages belong to the `Core` package. These are versatile tools that can be used in multiple analyzers and are, therefore, implemented in a more generic way. Specific algorithms, obviously, output less generic results, like the clustering algorithm discussed in section 5.3.3.

5.3.1 Core

The core package contains helper classes that assist more than one analyzer but can be made generic enough so that an analyzer specific implementation is not needed. A pure helper class is the `Operator` class, it gives us information about the transition between nodes. Traces only tell us an ordering that is observed, but that ordering is not necessarily the only one possible; we might observe both ABCD and ACBD. From this limited example, we might conclude

```
A -> (B -> C | C -> B) -> D
```

This cannot be represented using traces, therefore, we introduce Operators. Operators have a start event, an end event, a weight, and an operator type. The three operator types currently supported are `Sequence`, `ExclusiveChoice`, and `Parallel`. The Sequence simply indicates the transition from node start to node end. An ExclusiveChoice operator indicates that the transition is part of a group of ExclusiveChoice operators with the same start or end node, and only one of those operations can be chosen. The Parallel operator also indicates that it is part of a grouping, but instead of choosing one of the options, we now follow every path. The paths can be taken in any order and will most likely differ in multiple runs. These operators function similarly to the *cuts* in process trees [20], with the main difference being that we do not have a *loop* operator.

The previous example can now be represented by

```
Sequence(A, Sequence(Parallel(B,C), D))
```

K-successor algorithm

In order to transform the many thousands of traces into a dictionary of transitions, we implemented the k-successor algorithm. Currently, every analyzer implemented uses the k-successor output for further analysis, not the traces themselves, so you could argue that this is actually a parsing step. In effect, the distinction makes little difference. The algorithm transforms the individual event-instances contained in the traces into operations between the events in the list of events. In all present cases, the traces can be disposed to free up memory if needed after the k-successor algorithms has processed them.

The algorithm is initialized with a k-value; this determines the maximum look-ahead within a trace. First, an empty dictionary of dictionaries is created with the following internal properties:

```
Dictionary<int, Dictionary<int, Operator>>
```

The first key is the start node id, the second key is the end node id while the Operator defines the interaction that exists between them. This allows for intuitive access to operators with `Dictionary[Start][End]`, this would return the operator, or null if there is not connection. Traces are added to the list one by one by calling the code in Listing 5.2 for every trace.

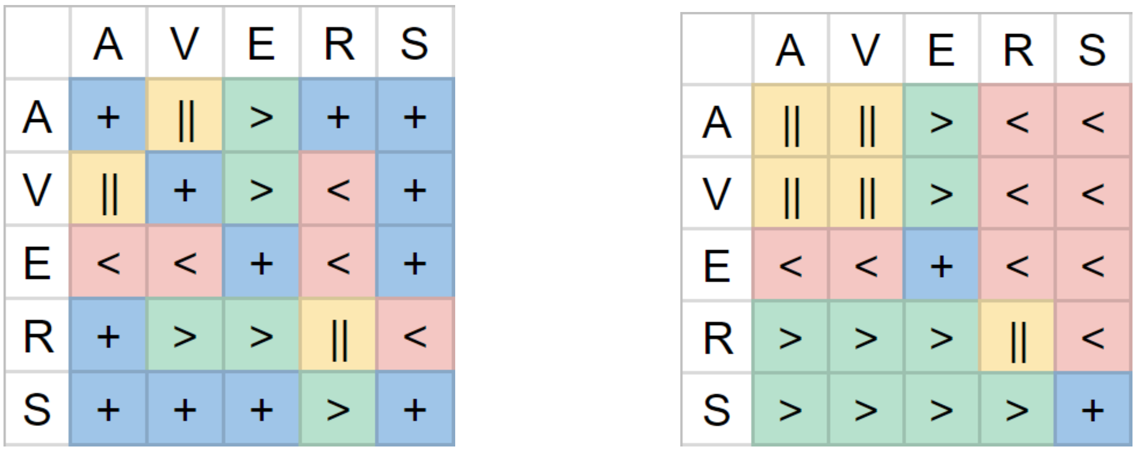
For each event, we look k events ahead in the trace and check if a relation between the events is already present in the dictionary. If it is not found, we check if the relation exists in the opposite direction. This would mean that it is a parallel operation, as both events occur but in arbitrary order. If a parallel relation is found, we insert it in both directions, the newly found with weight 1 and the other relation with its previous weight. If the relation already exists we simply increment

```
for (var i = 0; i < events.Count; i++){
    for (var j = i + 1; j < events.Count && j <= i + K; j++){
        var start = events[i].Event.ID, var end = events[j].Event.ID;
        if (!startDict.ContainsKey(end))
        {
            if (OperatorList.ContainsKey(end) &&
                OperatorList[end].ContainsKey(start))
            {
                OperatorList[end][start] = new
                    Parallel(OperatorList[end][start].Weight + 1);
                startDict[end] = Parallel(1);
            }
            else
                startDict[end] = new Sequence(1);
        }
        else startDict[end].Weight += 1;
    }
}
```

Listing 5.2: k-successor algorithm that is run for every trace

the weight by 1. If the relation does not exist in both directions we create a new sequence relation with weight 1.

When all traces are inserted we are left with a dictionary containing every transition possible within the system in k steps. The result of running the k-successor algorithm on Listing B.1 can be seen in Figure 5.3. The + symbol means the transition is not reachable, the > symbol means that the row event happens before the column event, while < is the reversed of >. The || symbol indicates the transition is observed in both directions.



	A	V	E	R	S
A	+	||	>	+	+
V	||	+	>	<	+
E	<	<	+	<	+
R	+	>	>	||	<
S	+	+	+	>	+

(a) k is one

	A	V	E	R	S
A	||	||	>	<	<
V	||	||	>	<	<
E	<	<	+	<	<
R	>	>	>	||	<
S	>	>	>	>	+

(b) k is infinite

Figure 5.3: A visualization of the output of the k-successor algorithm for varying values of k.

5.3.2 Flow diagram

We constructed a view that displays the same data as the matrix view, but in a more intuitive flow diagram. The only change in the data that differs from the matrix view is the addition of a START and an END node. The start node functions as a clear starting point and guarantees that all traces are part of the same web. The end node clearly displays what actions can be last in a sequence, as every action that does not link to the end node has to be followed by another action.

5.3.3 Clustering

The first parsing step created many event-instances, the k-successor algorithm extracts all possible transitions between events, but we are still left with an n^2 sized matrix for n events. Logs contain hundreds, sometimes thousands of unique events. The human brain cannot easily take in millions of interactions between components. Therefore, we extract the most meaningful transitions by clustering related components.

The AIM framework offers two types of clustering, one on a property of an event, and one on the fan in/out of the event. Both work with a depth parameter to determine the level of clustering.

Property

Property clustering merges events based on a predetermined property. Best results were obtained when clustering on the package property, as can be seen in Chapter 7.

A property only has a set amount of levels that can be clustered on, e.g. the package structure is often defined using multiple levels like `[runner.core.start]`. In the example of Listing B.1, we see a maximum of three levels, with the first level shared across all actions.

The algorithm is fairly straightforward, as can be seen in Listing 5.3: the groupings are all pre-computed as this requires only a LINQ GroupBy call. Simply merging the events, however, does not result in merged edges. When we choose what level merge is needed we create a `Cluster` of the grouping of nodes and update the fan in and fan out properties of the nodes contained in the cluster. This is repeated for each grouping.

```
PreCompute()
{
    foreach grouping-level in property
        events.GroupBy(property[grouping-level]);
}

GroupOnLevel(level)
{
    foreach grouping in groupings[level]
    {
        Cluster cluster = new Cluster(grouping);
        UpdateFanInOut(cluster);
        AllNodes.Remove(grouping.nodes);
        AllNodes.Add(cluster);
    }
}
```

Listing 5.3: Property clustering algorithm

Fan in/out

The fan in/out clustering identifies tightly coupled components and merges them into clusters. This is done by taking the most connected component in either fan in or fan. The fan count is the number of connections the component has to other components, either incoming of outgoing. This way, the highest amount of edges are eliminated per merge. The algorithm is, in essence, a greedy algorithm to produce a graph with the least amount of edges.

As there are always multiple candidates to match a node with, unless the highest fan count is equal to one in the entire graph, we need a way to find the second merge candidate. This is done by looking at the traffic over the edges; we find the component that has the most interaction with the first merge candidate.

After merging we update the fan in/out in the same way as with the property clustering.

Performance

Running time has not been an issue since the introduction of `Event, EventInstance` structure. When treating everything as an event memory usage on larger, 1GB, logs proves difficult for the test machine, since it only contained 8GB of RAM. It is essential that the entire event list fits in memory, as searches like in the Fan in/out clustering go over N/2 events K times where K is the merge depth and N the number of events.

Caching the fan in/out numbers of nodes was proposed, but as these numbers could potentially all be outdated after a single merge this was deemed inefficient. A sorted list may be kept and any changes made in updating the fan in/out when a new cluster is made would then be inserted in O(log n) time instead of the current O(1). This would save a single O(n) search for the node with the highest fan in/out, and does not guarantee increased performance.

5.4 Visualizer

The visual modules make the information from the analyzers human-readable. This will be the most flexible module in output types, as this is the end step in the SAR process. The ArchitectureCity visualizer, for instance, outputs a .scad file that can then be rendered using OpenSCAD, while the Petri net view writes the results to the Neo4j database and utilized the built-in graph rendering engine of the Neo4j dashboard.

SQ3. At what level of abstraction should we visualize the data?

The level of abstraction depends on the goal of the visualization; when identifying a single feature responsible for high CPU usage a detailed, low-level overview is needed while when listing what packages call a certain package a high-level overview is better suited.

As discussed in section 5.3.3, the clustering offers multiple levels of abstraction, giving the user the freedom to choose, or combine, views.

Using events and their ordering as the only input, effectively discarding all event instance information, does limit the level of detail. It is not possible to see all instances of a chosen event for further inspection, as there is no analyzer that supports such operations. This is, however, fully possible by designing an analyzer and visualizer that do offer such features, and is therefore not a weakness of the framework. The parsers are capable of passing all information contained in a log file, as they offer a `Blob` object that can be filled with any type of data. Currently, there is an unused feature as all relevant information of Case I and II fit in the basic data types of the `Event` class.

5.4.1 Matrix

The matrix output seen in Figure 5.3 can be considered a visualization but is not strictly implemented as such. The drawing functionality is contained in the k-successor class as it only outputs a CSV file with k-successor values, and has no visual component. The images seen in Figure 5.3 are made with Google Sheets conditional formatting and not a part of AIM. In the progress updates for Case I, however, it was found to have merit for software architects, outside of the intended debugging purposes. This is why this visualization was used throughout the case studies, as it offers a quick high-level relational mapping.

5.4.2 Flow diagram

To visualize the flows we chose a force-directed graph drawing algorithm. This algorithm is part of the Neo4j dashboard, therefore, inserting the nodes and edges into the database created the visualization. In Figure 5.4 we see our running example as visualized by the Neo4j dashboard.

The two edge colors represent the symbols from the matrix: the red arrows represent $||$ actions that can happen in any order, while the blue arrows represent sequences, $<$ and $>$ in the matrix. Arrows are directional and contain a number that corresponds to the number of times that path is taken. Every node, except the start and end, should, therefore, have an inflow equal to the outflow, this is clearly visible in Figure 5.4. At a glance, we can already see the flow of the system, as opposed to the matrix view that requires more time to comprehend.

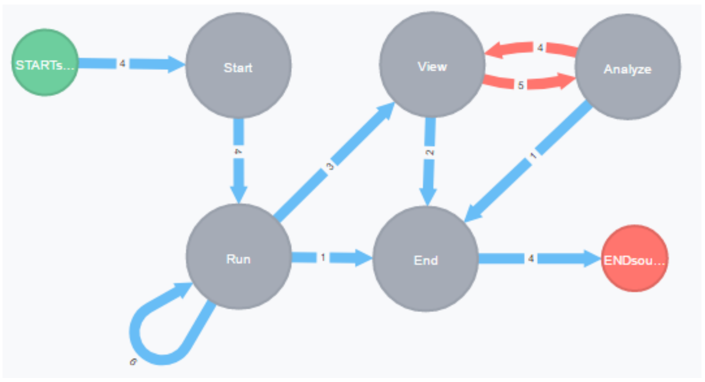


Figure 5.4: The flow visualization of the running example, manually rearranged for clarity. Blue arrows indicate a one-way sequence, while red arrows indicate a two way relation between the nodes.

Chapter 6

Visualizing Dynamics: ArchitectureCity

The most significant component of this thesis is the new ArchitectureCity visualization. It is a combination of a dependency graph and CodeCity, based on dynamic aspects of a running system. ArchitectureCity is implemented as a visualizer within the AIM framework and uses the output of the clustering analyzers as discussed in Section 5.3.3. CodeCity, as introduced in Section 3.4, is mostly used in combination with source code to show package structures and static information, e.g., LOC. A dependency graph usually gives no extra information regarding the nodes it displays, only the static interactions between them. Sometimes nodes are grouped in packages, both to be more visually appealing and to give additional context.

6.1 Global Idea

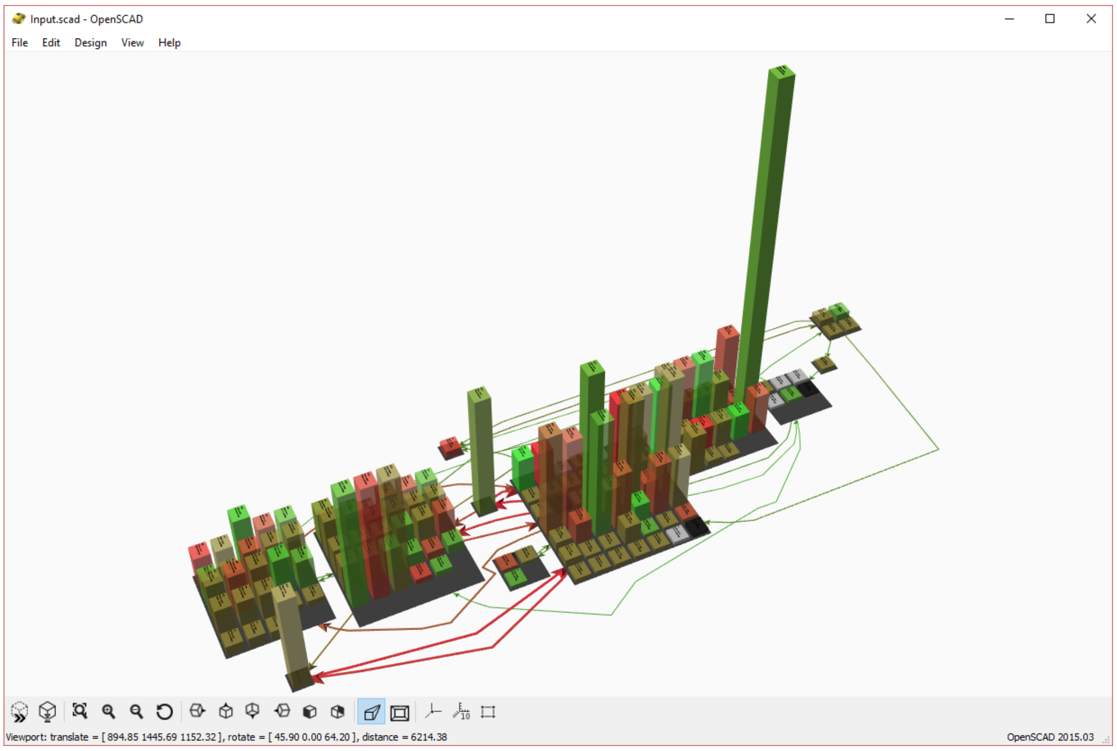
ArchitectureCity takes the high-level intuitive overview and node information properties of CodeCity and combines it with the interactions between components that we know from a dependency graph. All information is mined from logs, and is, therefore, dynamic, true system usage. The source code is not necessary for the creation of the visualization, and at this time, there is no method to enhance node information with static information, although this could be an interesting addition. Not requiring the source code is an advantage in and of itself, the source code could not be available anymore, or the company may not be willing to supply it for analysis. With ArchitectureCity we only need the log files to construct the visualizations.

Node information is displayed on the roads and buildings themselves by changing their appearances such as color and size.

6.2 Visualizing as a City

As with CodeCity, buildings are pieces of functionality, but as we have no knowledge of the actual source code, we create a building for each unique log event. This visualization is coupled with the clustering algorithms and they provide the city districts. A district is a cluster of buildings that belong to the same package, or have strong within-cluster interactions. The clustering algorithms allow for different clustering depths and, therefore, different levels of detail within ArchitectureCity. Each building belongs to a single district and each district contains one or more buildings.

The sizes and colors of buildings depend on various metrics of the events corresponding to the building. This allows us to display a range of metrics in a single view. As opposed to CodeCity, roads between districts are possible. These roads are directional and depict interactions between buildings from the connected districts. Relations are only represented between districts, not between buildings contained within a districts. An example of an ArchitectureCity visualization can be seen in Figure 6.1.



```
Viewport: translate = [ 894.85 1445.69 1152.32 ], rotate = [ 45.90 0.00 64.20 ], distance = 6214.38
```

Figure 6.1: A visualization of ArchitectureCity utilizing Package Clustering with package depth of 3.

6.3 Dimensions

ArchitectureCity is able to visualize many dimensions of information at the same time through different visual aids. Both buildings and roads have a placement, color and size, depending on the corresponding event or interaction.

Buildings

Buildings are the most flexible, as they possess a placement, height, width, length, color, and opacity. This would allow for six axes of information to be displayed at once. In its current state, ArchitectureCity buildings do not scale in width or length as this would require a new placement algorithm within districts and would probably decrease the clarity of the overall visualization. The height is based on the number of occurrences of the event within the log. As these values van range from one to tens of thousands, we chose the natural log of the number of occurrences as actual building height to reduce scaling issues. The colors can match the height of the buildings, or can show the fan in/out ratio. When displaying fan in/out, it uses a color range from green to red corresponding to a high in/out ratio and a high out/in ratio respectively. Buildings with only outgoing roads are light-gray while buildings with only incoming roads are black. The opacity displays the absolute fan in/out value, with a more transparent building having a lower absolute fan in/out count.

Roads

Roads do not posses as many dimensions but still feature placement, width, color, and opacity. The placement of roads indicates which components are connected, and in what direction communication takes place. The color and width depend on the strength of the interaction, in the current state it is based on the number of interactions. Alternatively, the roads can easily be modified to e.g. indicate the average time delay between actions or the volume of data passed between buildings.

6.4 Realization

The computations start with choosing and running a clustering algorithm to produce the districts. This information is fed into the Graphviz placement engine to compute the locations of each district and road. The districts are then populated with each individual building and its metrics. All information is then parsed to an OpenSCAD file that, together with Listing B.2, is able to be parsed into an interactive 3D environment using OpenSCAD.

Clustering

Districts depend on the clustering algorithm and level of depth chosen; a visualization with all buildings in the same district is possible, as well as every building in its own district. The former does not offer any special insight, but the latter shows a complete control flow of the software system. The two current clustering algorithms, Property clustering and Fan in/out clustering, are discussed in Section 5.3.3.

Placement

After completing the clustering the placement of the districts and roads is computed by Graphviz DOT, which generates a .dot file containing all information to create a 2D placement. This forms the 2D base for ArchitectureCity, as it uses these exact coordinates. As Graphviz clustering is an experimental feature that is not fully implemented, we opted to create the clustering ourselves, letting Graphviz generate a placement for the districts and roads and fill in the buildings afterward. The size of the district corresponds to the number of buildings it contains to always have enough space to later place them. The district width and length are always equal to the square root of the number of buildings it contains; this does mean that it is possible to construct a district with an empty row. This occurs when the number of buildings within a single district is n where $n \leq (\lceil\sqrt{n}\rceil - 1)\lceil\sqrt{n}\rceil$.

The output of the placement is a dot file containing all elements with their corresponding coordinates. This file is a valid .dot file and can, therefore, be used to generate a 2D image of ArchitectureCity.

Populating

The Graphviz coordinates and sizes are parsed to the general building and road classes. This intermediate step was chosen to facilitate the support of multiple rendering programs with minimal effort. The building class only has information about the shapes, colors and, sizes of the 3D object, as well as possible text to be printed on it. The roads class contains the buildings it connects, the intermediate points between the buildings as computed by Graphviz, and the color and width of the road. All colors and sizes of the buildings are computed in this step, as it is the step last in which we have access to all the event information. After this step, only the 3D object information is kept.

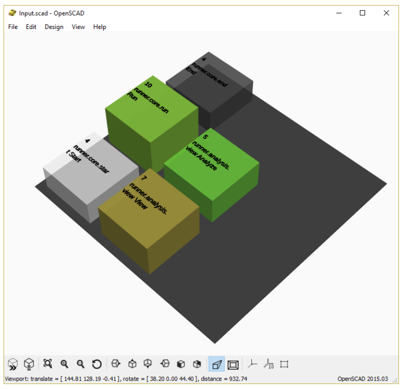
Rendering

After creating the buildings and roads, the visualizer parses the objects to the OpenSCAD format. The output file uses a helper .scad file that contains the logic for drawing the various objects, this file can be found in Listing B.2.

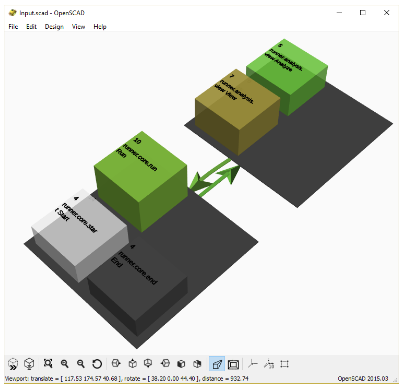
Anti-Aliasing (AA) can be added by flagging OpenSCAD for GPU assisted Anti-aliasing. To make the images more visually appealing, AA is applied in several of the ArchitectureCity examples in the cases of Chapter 7 and the running example in Section 6.5.

6.5 Running Example

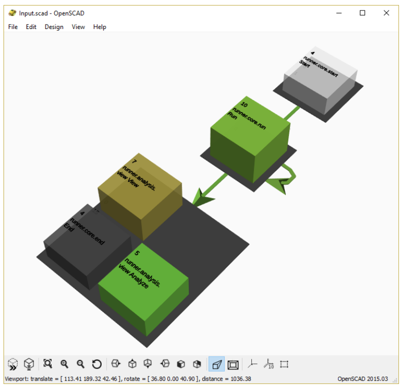
We see the ArchitectureCity of our running example in Figure 6.2. Because of the low number of uniquely logged functions, there are few buildings to be seen. ArchitectureCity is not very suitable for small systems as this one, even at the lowest level of abstraction. We see the package clustering, fan clustering and no clustering at all. Fan in/out clustering reduces the number of nodes to a specified max-Node value by clustering them. By supplying the algorithm with a larger max-Node value than the number of nodes present, it skips the clustering altogether, effectively returning the most detailed view. In Figure 6.2c we see that View, End, and Analyze are merged together. The rationale behind this can be extracted from Figure 6.2d; we start with clustering the black End node as it has the highest fan in/out, namely a fan in of three in. We look at the node with the highest fan in/out that feeds into the End node, which is View. After the merger the new merged node, Run, and Analyze all have the same fan in/out values and the first node in the list is chosen, this is Run, resulting in Figure 6.2c.


Input.scad - OpenSCAD
File Edit Design View Help
10 runner.core.run Run
4 runner.core.end End
4 runner.core.star t Start
5 runner.analysis. View Analyze
7 runner.analysis. View View
Viewport: translate = [144.81 128.19 -0.41], rotate = [38.20 0.00 44.40], distance = 932.74
OpenSCAD 2015.03

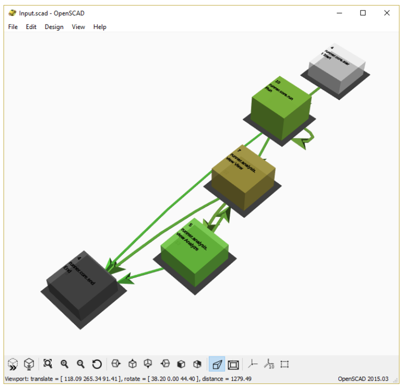
(a) Package Clustering with package depth of 1


Input.scad - OpenSCAD
File Edit Design View Help
Viewport: translate = [117.53 174.57 40.68], rotate = [38.20 0.00 44.40], distance = 932.74
OpenSCAD 2015.03

(b) Fan in/out clustering with 2 resulting clusters


Input.scad - OpenSCAD
File Edit Design View Help
Viewport: translate = [113.41 189.32 42.46], rotate = [36.80 0.00 40.90], distance = 1036.38
OpenSCAD 2015.03

(c) Fan in/out clustering with 3 resulting clusters


Input.scad - OpenSCAD
File Edit Design View Help
Viewport: translate = [118.09 265.34 91.41], rotate = [38.20 0.00 44.40], distance = 1279.49
OpenSCAD 2015.03

(d) No clustering

Figure 6.2: An ArchitectureCity visualization of the running example for varying clustering methods.

Chapter 7

Evaluation through a Multi-Case Study

To demonstrate the capabilities of the AIM framework, we present two cases from two different companies. The log structure and contents are widely different and, therefore, two parsers were constructed.

The difference in system architecture, or at least the constructed architecture, complement each other in showing the strengths and weaknesses of the framework.

The cases were selected without prior knowledge about their system or logging structure. The logs were available from the start of the construction of AIM, and have influenced some decisions by defining the data available for analysis. In the end, not all methods fit both cases equally well, but this serves well to illustrate the need for certain improvements to the framework.

We start with the first company, we'll call it DL, which handles delivery logistics in the Netherlands and has given us logs of six different systems, during three different time periods. The company of Case II, named FI, supplied a single log file containing information about multiple systems during a long period of time. FI is a very large financial company.

7.1 Case I - DL

DL has supplied log files of six different systems, back- and front-end, during three one-day intervals, so 18 files in total. The one-day intervals were chosen because the company mostly operates in a weekly cycle with a one day peak in processing. The logs are of the peak day when the system is most heavily utilized.

Parsing

An anonymized part of the log can be seen in Listing 7.1, all other systems logs follow the same basic structure. As can be seen, the structure is

```
datetime Type [package] (Thread) actionid
```

This makes it easy to parse using regular expressions: the first part is a datetime until the first space, then a logging type we will not use, followed by a package structure between [], then a thread id between () and the last part is the action id.

```
15-11-2016 03:30:05,032 TypeA [a.b.c.d] (T2) A1
15-11-2016 03:30:05,036 TypeA [a.b.c.d] (T2) A2
15-11-2016 03:36:05,009 TypeA [a.b.c.d] (T5) A1
15-11-2016 03:36:05,012 TypeA [a.b.c.d] (T5) A2
15-11-2016 03:42:05,008 TypeA [a.b.c.d] (T9) A1
15-11-2016 03:42:05,011 TypeA [a.b.c.d] (T9) A2
15-11-2016 03:48:05,009 TypeA [a.b.c.d] (T9) A1
```

Listing 7.1: Small part of an anonymized log of the DL company.

The biggest downside to the DL logs is the exclusion of a dedicated process id; the log only contains a thread id. From this, we assumed that all actions that run on the same thread belong to the same action, something we know is not true. As a remedy, we implemented a time-based feature that splits the thread separated actions into smaller chains of actions. It does so by introducing a split if there is at least a certain time between consecutively logged actions. This time is a variable that has to be set and is heavily dependent on the system itself.

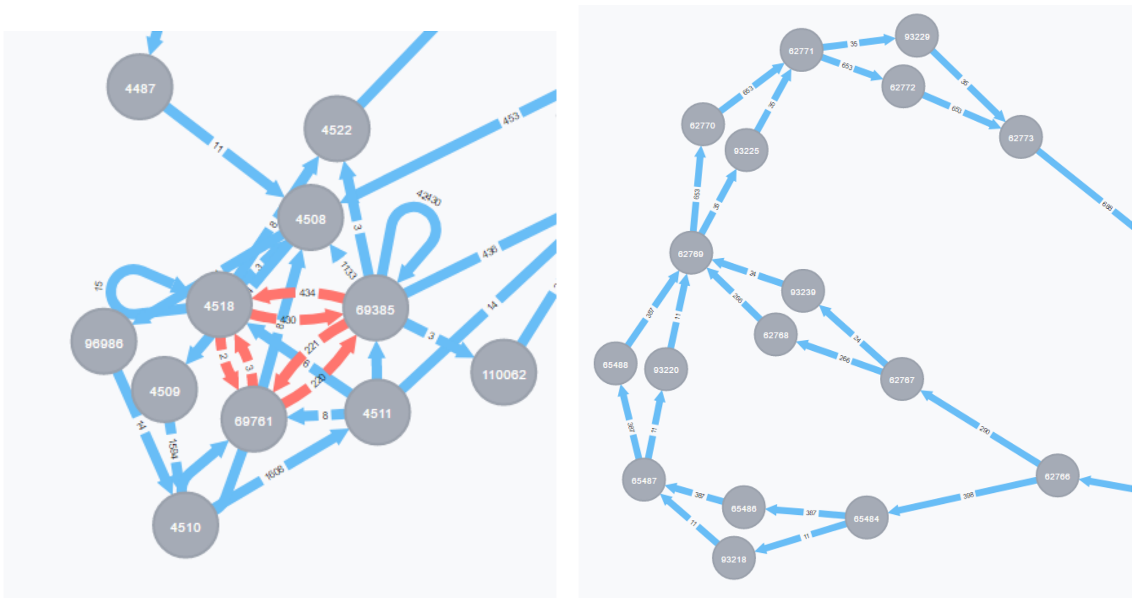
The running example uses the DL parser, as can be seen from the structure, as the DL logs are easy to read and do not contain many fields.

Visualizing

The matrix view for $k = infinity$ of Back-end-1 can be seen in Figure A.7. We can clearly see the package structure, even without the gray-white indications. We also see one package at the center that has no interaction with other components and has a clear ordering, indicated by the absence of yellow both-directions squares. This indicates that this is a separate component with a linear structure. The component stays separated from the others even if we turn the time interval split off as we can see in Figure A.9. We do see something interesting when no interval splits are active; the whole package turns yellow. This indicates that this process is run again in the next time interval, more than 22 hours after the previous as we can see in Figure A.8.

We identify more linear packages, and two highly flexible packages containing many functions. Even there we identify ordering with regards to other packages; the top-left yellow cluster follows the first two packages on the top rows while the bottom-right yellow cluster comes after the package above it, colored in red.

An overview of the workflow net of the Back-end-1 system can be seen in Figure A.10. Although it looks like a spaghetti network at the center, we still see clear control flow patterns. In Figure 7.1 we see a few interesting places within the workflow. Figure 7.1a shows a self-edge with an occurrence count of 42430, about ten times higher than the second highest in the graph. Figure 7.1b shows a clearly ordered structure with several paths that all come back to the same action. This shows us exclusive choices being made within the system. Looking closer at the edge counts,



(a) A self edge with 42430 calls.

(b) A clear structure with choices.

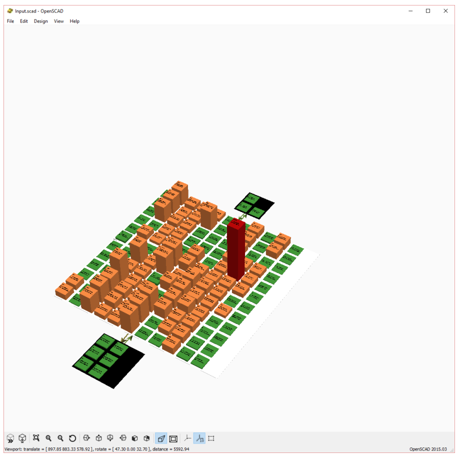
Figure 7.1: Zoomed in sections of Figure A.10.

we see that the path from node 62766 through 65487 to 62769 has two choices. In these two choices, both times one path is taken 11 times while the other is taken 367 times. This suggests that one variable is tied to both choices in the sequence and that the 11 that go through 93218 also go through 93220, but we cannot be certain of that.

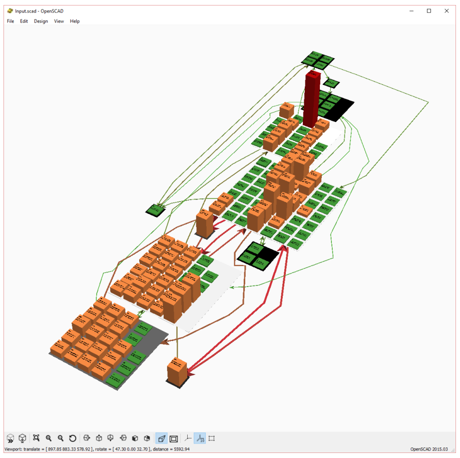
The ArchitectureCity overview gets most of its information from the data behind the matrix visualization but creates a more intuitive overview. We modeled the same log file with the 30-minute split-time with package level clustering on 1, 3, 5, and 7 as can be seen in Figure 7.2.

We see the unconnected package from the previous section appear in Figure 7.2c in the top right corner as an unconnected black square. In the lower package depths it was still part of a bigger package and seemed connected to the rest of the system, from this we can conclude that it is important to not rely on a single level of depth. We clearly see what activities are called more frequently by the height and color coding. Interaction frequency is also easy to perceive as the road width, and color correspond to the interactions made.

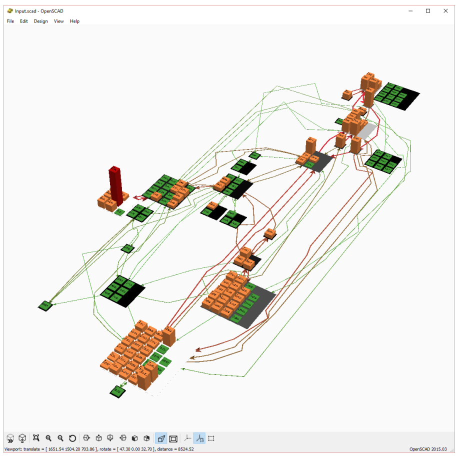
From Figure 7.2a we can conclude that there is little interaction between top-level packages, as is to be expected. Two layers down in Figure 7.2b we see more packages being split, creating a few single-action districts. More interactions become visible, but a clear structure remains present. This trend continues with level five and seven, with a linear increase in both districts and roads, as can be seen in Figure 7.3. The same trend holds for other subsystems of DL as can be seen in Figure A.11. In this front-end subsystem, we see two unconnected components through the fan in/out clustering. The system expands exactly as expected with the smaller system first, as it has the fewest connections. With the 25 clusters, we see in Figure A.12b that we can already conclude that the smaller system is purely linear and that the larger system also contains a purely linear component. At 50 clusters we notice that there are two large multi-event clusters, which is unlikely to happen in the fan in/out clustering. As expected, the smaller event cluster is split up first as we see in Figure A.12d. In Figure A.13 we see the entire system unclustered, showing us multiple linear processes, a few entry points of the application in light gray buildings, and a few exit points shown in black buildings. We do see many bold, red lines indicating heavy traffic between components. Although the linear paths seem well traveled, the deviations from those paths are also widely used.



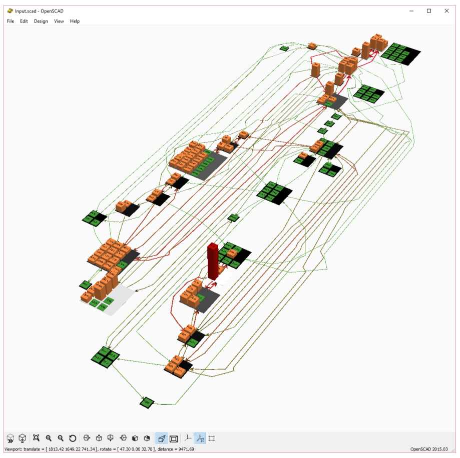
(a) Package Clustering with depth of 1



(b) Package Clustering with depth of 3



(c) Package Clustering with depth of 5



(d) Package Clustering with depth of 7

Figure 7.2: A visualization of ArchitectureCity for varying levels of package clustering. Logs are from Back-end-1 using call count based coloring.

Evaluation Case I

During the evaluation, we discussed the merits of the AIM framework, and ArchitectureCity in particular, with the software architects responsible for the DL software system. The first visualization shown was an ArchitectureCity representation containing a single high peak in an otherwise low-rise environment. The peak was immediately identified and linked to an action in the correct sub-system, without any further information about which one of the six sub-systems was visualized. Of the two logs presented, both visualizations were correctly matched to the logs with just a high-level overview of ArchitectureCity, package level three. Most components could be guessed correctly from the overview. As multiple levels of clustering were discussed, the question was raised what levels would be most useful. For the fan clustering, higher levels tend to be more effective as the lower levels tend to contain semi-clustered processes where a part is clustered and another part is not. These semi-clustered processes make the visualization more difficult to comprehend.

Clear benefits of the system were identified immediately. Most levels of depth have their merits, as the usage will depend on the goal of the user. Therefore, the fan clustering with infinity as max cluster count is also useful, as this shows a unclustered view of the application but still retains the

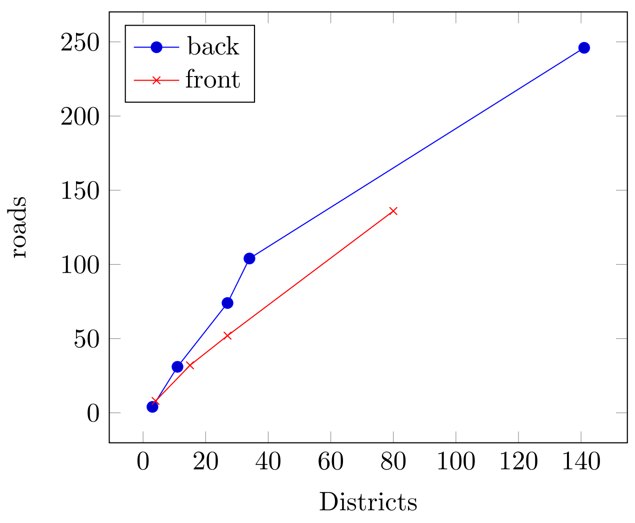


Figure 7.3: The ratio of roads and districts for different cluster depths for the front-core and Back-end-1 subsystems.

placement engine behind ArchitectureCity to group components. A split out view still contains useful information regarding individual events, as their frequency and connectivity can be studied. In one case there was a strange, above-average activity node that was inspected in more detail and the event information surprised the architects. This component was thought to be removed or de-activated but was still showing up in the logs as an active component. Insights such as these were deemed incredibly advantageous, and the model was called a "model of truths" instead of the intended models on which the application was based. All systems shown were correctly identified, and clusters and system flow were largely expected and true to specification. They were surprised by the amount of information their log contained and happily surprised by the visualization of it.

Some improvements were requested: a tool-tip, more information on the clusters and making buildings clickable to their source. Adding time-based features was also discussed. The generation of interval based slices of the log and visualizing each one yields a "living" 3D overview of real system usage. This was recognized as a fascinating feature that would only be possible through processing dynamic system information and, therefore, impossible to achieve using their current techniques.

The correct conclusion that the system was written in Java was drawn purely based on the package level clustering visualization of ArchitectureCity.

The matrix view was also discussed, as well as the time interval to split traces. No consensus was reached on the interval, as this number would depend on too many factors and could possibly be different for every function. The matrix view was seen as a quick overview of function interaction for programmers and higher level stakeholders alike.

In conclusion, the visualizations shown exceeded the expectations of the architects, and many insights e.g. the thought to be removed component, sparked interesting discussions about the DL system. ArchitectureCity improved their understanding and insight into their own software with which they have had years of experience.

7.2 Case II - FI

FI initially supplied a complete log of their entire system, but as performance and the clarity of the visualizations were lacking due to the size of the system a down-scaled version containing only two sub-systems was supplied. This log was used in all visualizations and evaluation below and was picked with a special goal in mind. FI wanted to know if there was a difference in architectural clarity between two existing systems, one old system that has a lot of technical debt and a newer system that has, in their opinion, a clearer architecture with less technical debt. The log has the two system intertwined, without special indication of what part of the log belongs to what system. Between the systems, there are close to 55.000 lines of logging. The two systems do not interact so distinguishing between them is not a difficult task using fan clustering.

Parsing

When running the analysis, we found many actions that could occur in any order, resulting in many edges between nodes. We also identified a set of nodes that were incident with over 50% of the edges, while only representing less than 5% of the nodes. Another problem was that there were many events with only subtle differences, e.g. a multitude of SQL load statements, that, due to concurrency, were resulting in many edges. To tackle these issues we compressed and pruned the event space.

The events with subtle differences were merged with each other. This meant that the number of event-instances of a group of actions was preserved. Edges that would normally connect to many of the subgroups within the new cluster would effectively be bundled into a single edge. It also reduced the number of events and, therefore, the number of nodes in the visualizations. The number of nodes was reduced from 490 to 161, a 67% reduction, while the number of edges was cut from 4117 to 862, an 81% reduction. By pruning events entirely we reduced the number of event instances from 54492 to 35281, a reduction of 35%. In Figure 7.4 we see the number of event instances, events, and edges for the original dataset, the merged set, and the pruned set. Even after pruning the dataset, the network is heavily connected with a node having, on average, 10.7 in- or outgoing edges.

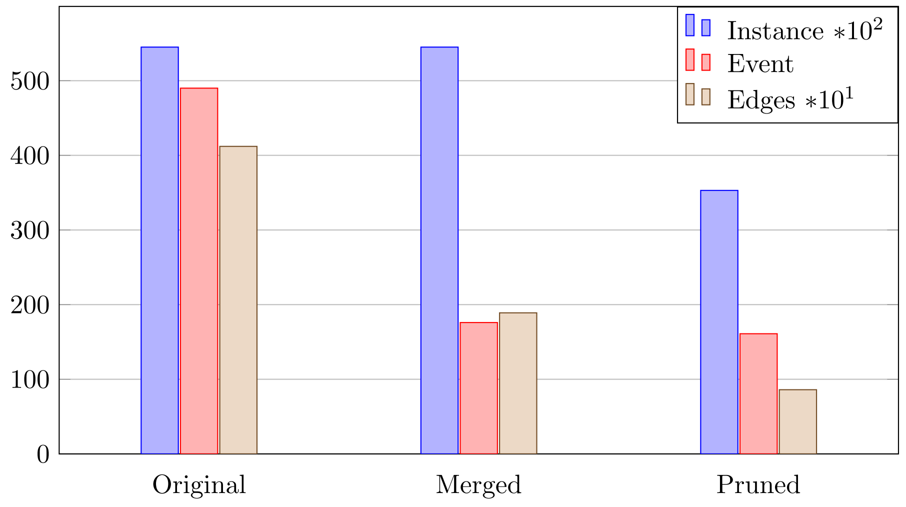
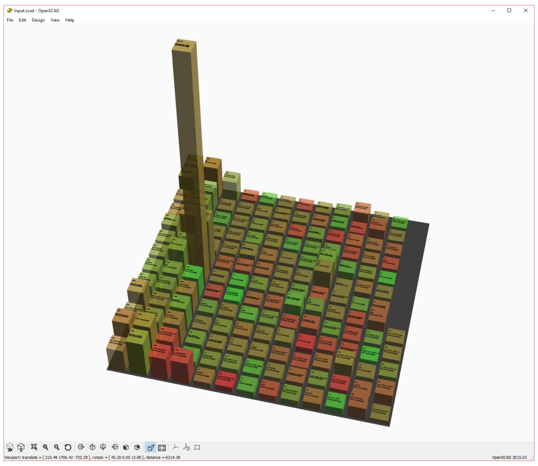


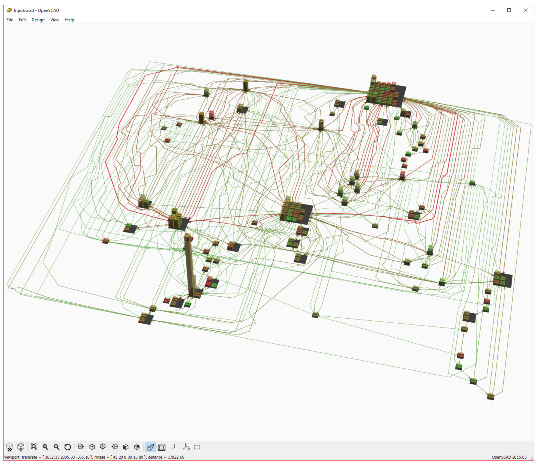
Figure 7.4: The number of event instances, events, and edges for the original FI log, merged log, and the pruned log.

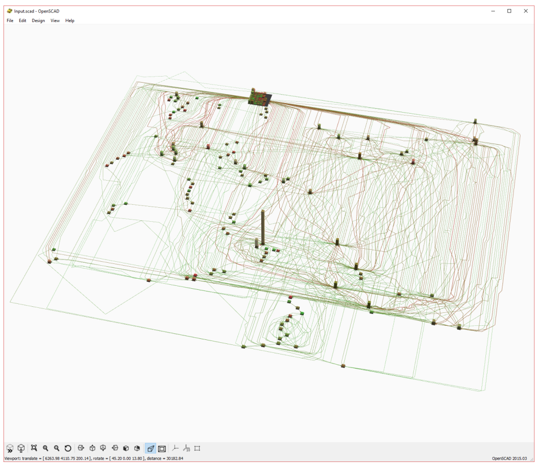
To select the events that would be merged or pruned, we used the fan in/out clustering algorithm to find the most connected components within the system. If these were specific to the system they were merged but if the event was a generic one e.g. “Execute SQL query” they would be removed. This method ensured that integral system components would still be present in the final visualization and that components that produced the highest number of edges would be pruned or merged first.

Visualization

We see the result of utilizing the pruned log in Figure 7.5. The log only contains three levels of package depth, with level one being the same in all events. This means that level two is already largly spread out, and we see many single-event clusters. We also see that the two systems use the same package names, as it is impossible to distinguish the two systems, even at the deepest package level. We see that package level three with pruning looks similar to level two without pruning, but with very little clusters left. The edges are more readable, and as only generic events e.g. SQL queries, retrieving IDs, start-log, and end-log were pruned, the custom system components and their flow remain mostly intact.


(a) Package Clustering with depth of 1


(b) Package Clustering with depth of 2


(c) Package Clustering with depth of 3

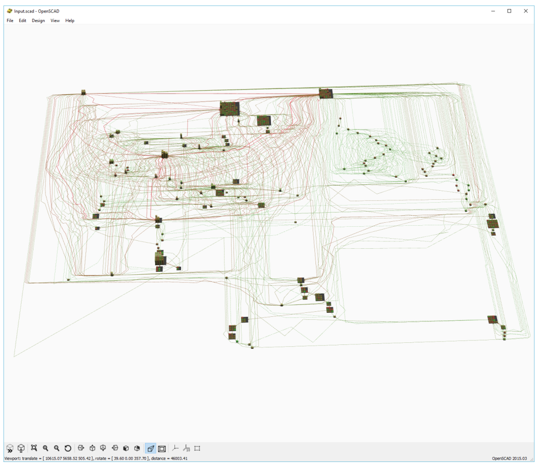
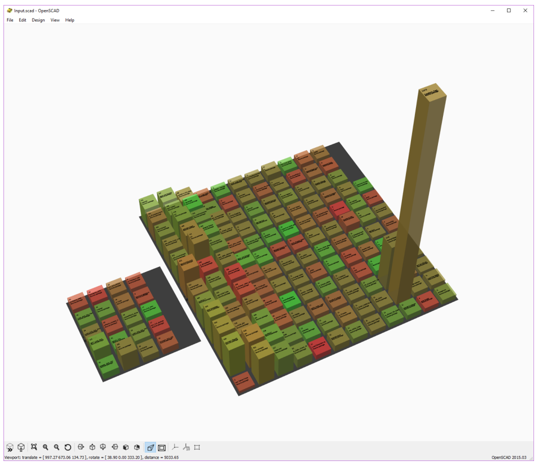

(d) Package Clustering with depth of 2 without pruning

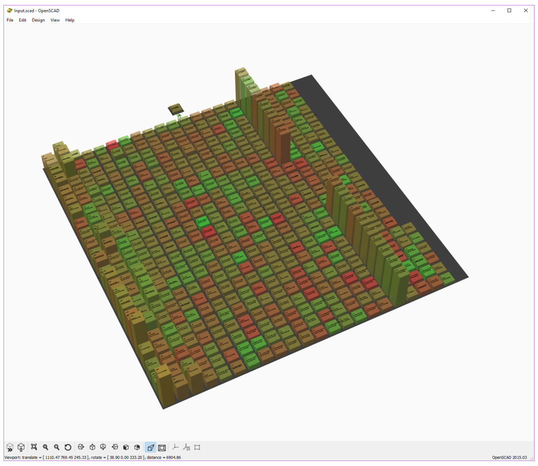
Figure 7.5: A visualization of ArchitectureCity for every level of package clustering, and level two clustering without pruning. Larger versions are available in Figure A.14, A.15, and A.16.

With fan in/out clustering, non-interacting components cannot be clustered, as there is not a single connection between them. In Figure 7.5 we see that pruning the events has also removed the bonds between the two separate systems. This was possible because the pruning focused on removing generic components that may be called from many classes. It was found that the log entry “Sequence Container” was the connecting component logged by both systems. This is clearly a generic term that does not imply any interactions or shared components between systems, as this is the only entry found in both systems.

Figure A.19 and A.20 indicate that the system might contain components that can be executed concurrently, as most processes are found to be executed in arbitrary order. The two systems are visibly divided, with “Sequence Container” being the only event that has interactions with both systems. The image indicates that concurrency is heavily used in both systems, with the bigger, old system showing signs of at least partial ordering within components as shown in Figure A.19. This indication of concurrency would also explain the chaotic nature of the ArchitectureCity. As of now, ArchitectureCity has no efficient methods of dealing with concurrent behavior. Worth noting is that the newer system has very high reachability between nodes, as almost all actions can be reached from any node.



(a) Fan clustering with 2 clusters with pruning



(b) Fan clustering with 2 clusters without pruning

Figure 7.6: A visualization of ArchitectureCity for fan clustering with 2 clusters, with and without pruning

Evaluation Case II

The goal of FI was to visualize the difference between the old and the new system. To achieve this goal we needed to separate the systems in the visualization, this was done with the fan in/out clustering. Unfortunately, the package clustering was not suitable to separate the systems, and the visualizations that resulted from it were deemed inferior to the fan in/out clustering. The absence of clear clusters with lower outside interaction indicates low cohesion on a package level.

The package clustering visualizations, Figure A.14, A.15, and A.16, did produce an interesting insight; it was correctly observed that the system of FI was built in C#, without any prior knowledge about the software. C# often has a lower maximum package depth than e.g. a Java system, and will more usually yield results similar as those observed in this case.

The fan in/out clustering visualization of ArchitectureCity was shown with and without pruning, to demonstrate the value of clustering and removing nodes to increase visibility. It was agreed upon that the pruning step increased clarity and did not remove essential components. Especially the fact that it resulted in the separation between the systems was seen as very valuable.

When step-by-step increasing the cluster count for the fan in/out ArchitectureCity we saw the newer system open up first, indicating that the fan counts for the components were lower than counts for the older system. Lower fan counts are an indicator for loose coupling as interactions between components require knowledge of the component. The newer system did, in fact, perform better than the old system, but as it is also smaller: a lower fan count was to be expected. When every node was separated, as seen in Figure A.21, a linear path was observed in parts of the system e.g. in the lower left. It was confirmed that the many edges still present in such a linear process were due to multithreading. It was confirmed that sequences would require multiple actions to be taken, but in arbitrary order with the level of concurrency depending on the server load at that time. Unfortunately, ArchitectureCity does not catch such behavior in a visually appealing manner but displays all the possible paths. It was agreed upon that heavy concurrency was the main problem in the visualization.

In Figure A.17 we see the flow of data in the old FI system, as indicated by the many arrows to the actions in pink. These processes possess many possible inbound paths and would, therefore, show up as such in ArchitectureCity.

In Figure A.20 we see the data flow of the new FI system. It seems to be more linear than the ArchitectureCity implies, however, within these steps, there are sub-steps that can be executed in parallel. This is visible when we look at the newer system without any clustering or pruning as in Figure A.22. Most of the spaghetti-like interactions are within the same package and are almost entirely caused by concurrent execution of code.

In conclusion, the goal to differentiate the two systems was achieved, but compared to the DL case the insights were of lower quality. Because of the concurrent nature of the applications, the output was highly connected and revealed a spaghetti-like dependency graph. The newer system did seem to be more loosely coupled, and have a higher cohesion than the older system. Unfortunately, due to the high complexity of the ArchitectureCity models, immediate feature identification was tougher than in the DL case. Connected components could be predicted correctly most of the time, and the two systems were unmistakably different.

7.3 Discussion and Limitations

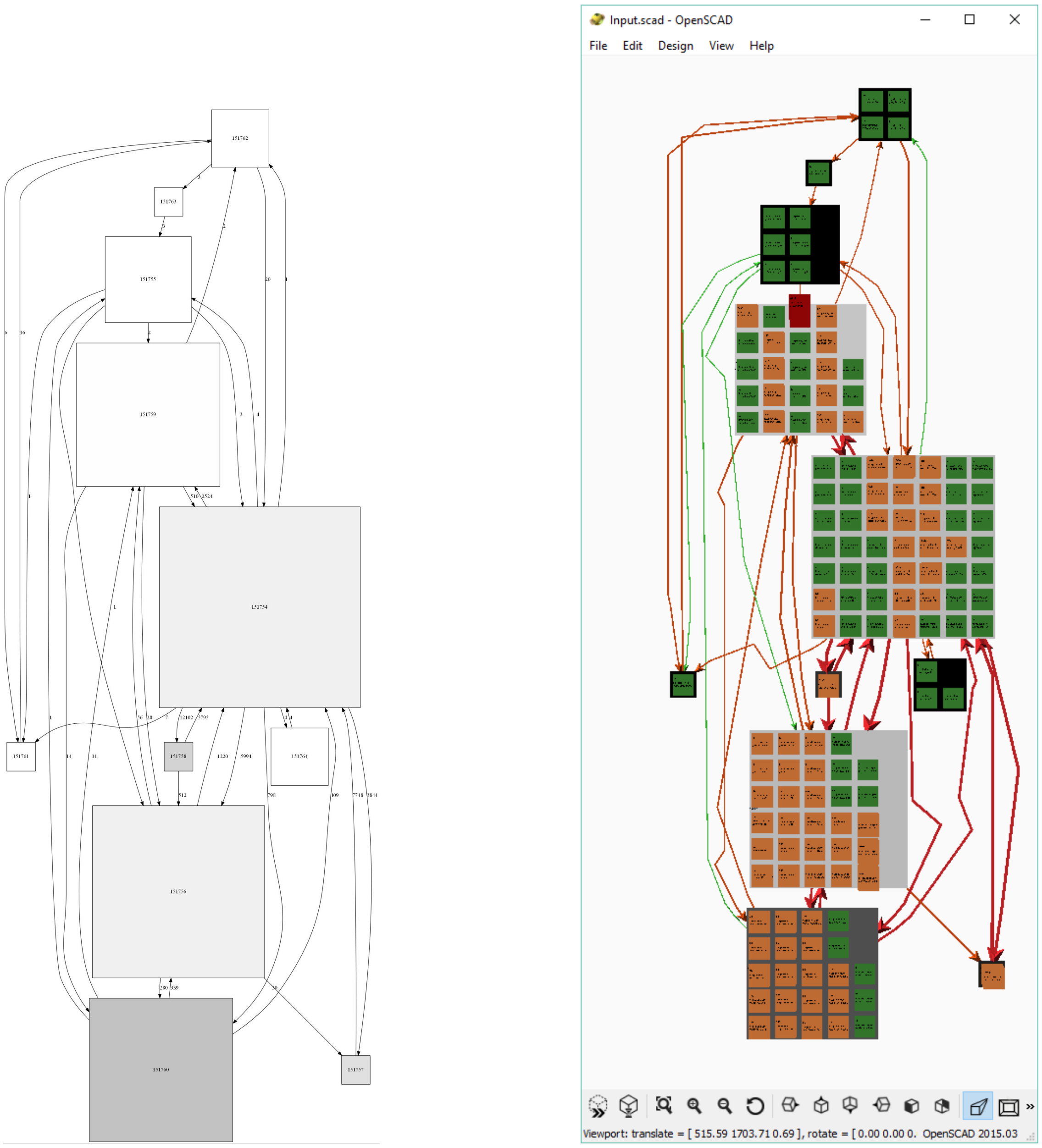
The visualizations suffer from the garbage in, garbage out principle, meaning that low quality logs will produce low quality visualizations. This is true for the AIM framework as it interprets the logs as absolute truths.

Some limitations are immediately visible when dealing with high road counts; roads become chaotic and increasingly difficult to interpret. Part of the problem is the way Graphviz computes the coordinates for the roads. Graphviz uses bezier curves to connect the points, while we simply draw a straight line between them. This is mainly because OpenSCAD does not support bezier curves, and implementing such a feature in a 3D environment is not within the scope of this thesis. This does mean that lines sometimes overlap where this is not necessary. The difference between the Graphviz drawing and the OpenSCAD visualization can be seen in Figure 7.7; the 2D drawing created by Graphviz from the exact same coordinates gives a superior result.

When graphs become very large, with edge counts in the thousands, Graphviz has trouble rendering the result. Our tool, however, has had no problems in rendering although memory usage may become quite high.

In theory, Graphviz has support for sub-graphs which would be well suited for our clusters. In practice, however, the support seemed to be lacking. Layouts become sub-par when this feature is enabled, especially within clusters themselves.

Another option was using Graphviz to compute the clusters as separate entities and then stitch them together to create the city. No suitable Graphviz rendering mode gave a placement that was orderly enough, while still maintaining exact sizes, to still capture the city like feel. The Patchwork mode seemed promising but did not follow exact dimensions and, therefore, gave wrong impressions of the data. Different modes can be seen in Figure A.3 through A.6. We see the Graphviz Dot algorithm performing poorly in the layout of its inner components, as it simply lists them from left to right. This results in incredibly stretched out visualizations that are hard to comprehend. The edges are handled fairly acceptably as they run from package to package, something no other algorithm seemed to support. The fdp algorithm had better performance


151762
151763
151755
151759
151754
151761
151758
151764
151756
151760
151757
Input.scad - OpenSCAD
File Edit Design View Help
Viewport: translate = [515.59 1703.71 0.69], rotate = [0.00 0.00 0. OpenSCAD 2015.03

(a) 2D Graphviz drawing using bezier curves

(b) 3D ArchitectureCity, a top view using straight lines

Figure 7.7: A visualization of the same architecture from a 2D and 3D view to illustrate the difference in edge clarity.

regarding the sub-graphs but seems quite random. The edges run from a random building within a package, instead of the package edge, as well as ignoring collisions by running in a straight line from building to building. Next, neato ignored sub-packages while still drawing them in some sort of overlay. This algorithm clearly is not designed with sub packages in mind. Lastly, patchwork ignored sub-packages and creates a square with all buildings patched together. In all previous algorithms, the buildings at least retained their true size, however, with the patchwork algorithm, some get stretched out to fit them into the square.

Chapter 8

Conclusions and Future Research

Software operation data captures the actual behavior of software. In this paper, we have presented an approach that takes this data as input for architecture mining.

Our research question as formulated in Chapter 2 reads

RQ. How can we capture dynamic aspects of running systems to construct relevant architectural views and perspectives for stakeholders?

To answer our research question, sub-questions where formulated that each address a part of the main research question. We will fist answer each sub-question individually before answering the main research question. Each sub-question was already addressed in the chapter it was introduced.

SQ0. What are relevant architectural views to capture dynamic aspects of a system?

As stated in Section 3.1, relevant architectural views to capture dynamic aspects of a system are sets of system elements and relations associated with them, that can be mapped to an architectural concern of one or more stakeholders and visualizes the behavior of an example run of the system.

SQ1. What is the current state of the art in software architecture reconstruction with respect to dynamic aspects?

In Section 3.2 we see a taxonomy of the current SAR landscape. The broad field was divided along five axes; goals, processes, inputs, techniques and outputs. Each of these axes features a variety of possibilities. Combining the axes gives us a method of describing each current SAR application.

SQ2. How to extract dynamic information from a running system?

In Chapter 4, we see multiple methods of extracting dynamic information from a running system. Most commonly, logging data is used as the information source regarding dynamic information. These logs can be audit logs, event logs that many systems produce. We set two requirements for the log data, the presence of an event identifier and an ordering of events. When the logging of the application is insufficient, we can inject logging statements into the byte-code of an existing system. This gives software systems that lack the level of logging needed for process mining a non intrusive method of producing higher quality and consistent logs.

SQ3. At what level of abstraction should we visualize the data?

The level of abstraction depends on the goal of the visualization; when identifying a single feature responsible for high CPU usage a detailed, low-level overview is needed while when listing what packages call a certain package a high-level overview is better suited.

We proposed a plethora of methods to visualize data at multiple levels of abstraction. ArchitectureCity offers the flexibility to let the user decide the abstraction level and make adjustments based on the visualization of the data itself.

SQ4. Can we extract the level of abstraction at which we should visualize the data from the data itself?

The data offers multiple axes for clustering within ArchitectureCity, as we have seen in the clustering algorithms. For instance, the package level clustering offers different ranges of abstraction based on the information present in the data.

We have developed the AIM, and more specifically, ArchitectureCity, which uses the analogy of cities to visualize the runtime of software: buildings, representing individual architectural elements are grouped in districts based on different clustering techniques, and streets depict the traffic between the different districts. Additionally, the streets, buildings, and districts can be coloured according to different metrics.

In conclusion, we are able to capture dynamic aspects of running systems to construct relevant architectural views and perspectives for stakeholders by analyzing the logging data the system produces. Multiple levels of abstraction can be applied depending on the stakeholders, where the ranges of possible visualizations are contained in the data itself.

Initial validation through a case study shows the potential of the proposed approach: we provided the architects with new insights on the operation of their software.

Future research

We see tremendous potential in ArchitectureCity because of the many additions that can be made to the software. First of all, the metrics it displays on the buildings and roads can be extended to feature performance data. This would mean that not only the flow of the application is visible, but also the bottlenecks within the software. As all information is purely based on log data, this would yield performance metrics as observed from real system usage.

We also see potential in adding a static code component to enrich the data displayed in ArchitectureCity. When coupled with feature location and tooltip support, this could be a useful feature for programmers, as the high-level overview can be directly translated to the low-level components and structures. Developers would immediately be able to view the complexity of the events logged and displayed in ArchitectureCity in terms of e.g. lines of code.

Matching the features to the code, one could also easily identify potentially dead code. As we link every function or class to a log entry, it would be trivial to compare the list of all events to the list of events logged by the system. Any code not reached could be displayed and flagged for inspection.

Another relevant addition would be adding a fourth dimension: time. As all data is already available in the logs, adding this feature would mostly consist of implementing the complex visualizations it would require. We envision a pre-computed series of ArchitectureCity 3D environments that are based on time slices. It should be possible to move through these slices to see components interact with each other during a certain time-frame. This would allow for performance inspection of the entire system in an intuitive 3D environment.

Another variation on the time-based approach would be the morphing code evolution. Instead of taking the same system at different time interval, we take various revisions of a system to see the evolution of system over time. This allows for in depth evolution views using real system event data. The impact of features would be easily analyzable, especially when combined with the previously mentioned performance data inclusion.

The component that causes the most inconvenience both in terms of performance, as well as feature restrictions, is the Graphviz layout engine. When parsing larger systems of over 500 nodes and over 2000 edges, Graphviz becomes unusable due to memory limits in Graphviz itself. The node placement also takes over 90% of the entire computation time, excluding the 3D rendering.

In theory, Graphviz supports sub-graphs and would be very suitable for our districts. In practice, however, this was found not to be the case. Sub-graphs are only supported by some dialects of Graphviz, and those do not handle the roads in the desired fashion. This is why we chose for DOT, and handled the inner components of a district ourselves. Ideally, the placement engine would handle all buildings and roads. As the quality of the placement was very high, we stayed with Graphviz for ArchitectureCity.

Graphviz edges, what we call roads, output a series of points that form a bezier curve. OpenSCAD, however, does not include bezier curve functionality and we simply connect the series of points. This creates sub-par quality roads as they regularly cross in cases where the bezier curves would not. This also causes some roads to seem to connect with buildings that they have no true connection to.

Another problem with OpenSCAD is the memory limit. We already had visualizations that got just over the RAM limit OpenSCAD employs with the Case II package level 3 un-pruned ArchitectureCity. Ideally, we would like to see a web interface that draws ArchitectureCity, as OpenSCAD does not support additional features like tooltips as it is designed for designs of physical products. A native HTML5 solution would also decrease dependencies on external software, therefore, increasing robustness and making the installation and deployment of the system more straightforward. It would also facilitate the incorporation of ArchitectureCity in a web-based dashboard, as is the norm with many code quality tools.

The last concern is the handling of multi-threaded processes in ArchitectureCity. As we have seen with FI in Chapter 7.2, multi-threaded application cause edges to be drawn for every possible ordering of events. We would like to see a form of grouping for parallel processes that eliminate the need for edges between them and bundle incoming edges. It would reduce the number of edges, effectively mitigating the performance issues of Graphviz and OpenSCAD. This would also allow ArchitectureCity to visualize exceedingly large systems while still retaining the clear, high-level overview.

Bibliography

[1] CodeCity by R. Wettel. `http://wettel.github.io/codecity-wof.html`. Accessed: 2017-04-08. ix, 16

[2] D3.js. `https://d3js.org/`. Accessed: 2016-11-16. ix, 17

[3] Logentries. `https://logentries.com/`. Accessed: 2016-10-31. 14

[4] Loggly. `https://www.loggly.com/`. Accessed: 2016-10-31. x, 14, 58

[5] Scalyr. `https://www.scalyr.com/`. Accessed: 2016-10-31. 14

[6] W. van der Aalst et al. Process mining manifesto. In *Business process management workshops*, pages 169–194. Springer, 2012. 21

[7] W. van der Aalst et al. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012. 10, 25

[8] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, 3 edition, 2012. 1, 7, 11

[9] G. Bastide et al. Adaptation of monolithic software components by their transformation into composite configurations based on refactoring. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4063 LNCS:368–375, 2006. 2

[10] J. Buckley et al. Real-time reflexion modelling in architecture reconciliation: A multi case study. *Information and Software Technology*, 61:107–123, 2015. 1

[11] P. Clement et al. Documenting Software Architectures: Views and Beyond. SEI Series in Software Engineering. (6):740–741, 2002. 7, 8

[12] M.E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968. 11

[13] B. F. van Dongen, A. K. Alves De Medeiros, and L. Wen. Process mining: Overview and outlook of Petri net discovery algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5460 LNCS:225–242, 2009. 15

[14] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, jul 2009. ix, 1, 7, 9, 10, 11, 12

[15] J. Gantz and Reinsel D. THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digi tal Shadows, and Biggest Growth in the Far East. *Idc*, 2007(December 2012):1–16, 2012. 14

[16] C.W. Günther and E. Verbeek. Xes standard definition. *Fluxicon Process Laboratories*, 13:14, 2009. ix, 21

[17] K. van Hee, O. Oanea, R. Post, L. Somers, and J.M.E.M van der Werf. Yasper: a tool for workflow modeling and analysis. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 279–282. IEEE, 2006. ix, 15

[18] G. Huang et al. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, 13(2):257–281, 2006. 10

[19] B.J. Jansen. The Methodology of Search Log Analysis. *Handbook of Research on Web Log Analysis*, pages 100–123, 2009. 13

[20] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs-a constructive approach. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 311–329. Springer, 2013. 28

[21] W. Löwe et al. Understanding software-static and dynamic aspects. In *17th International Conference on Advanced Science and Technology*, pages 52–57, 2001. 9

[22] M.W. Maier et al. Software architecture: Introducing IEEE standard 1471. *Computer*, 34(4):107–109, 2001. 8

[23] McKinsey & Company. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, (June):156, 2011. 14

[24] J. Peters and J.M.E.M van der Werf. A Genetic Approach to Architectural Pattern Discovery. *ECSAW*, '16 Proccedings of the 10th European Conference on Software Architecture Workshops, 2016. 1

[25] I. Raedts et al. Transformation of BPMN Models for Behaviour Analysis. *Msvveis*, pages 126–137, 2007. 15

[26] N. Rozanski and E. Woods. Applying Viewpoints and Views to Software Architecture. *Computer methods and programs in biomedicine*, (2005):11, 2011. 8

[27] J. Scheffer. Dealing with missing data. 2002. 14

[28] H. van der Schuur et al. Reducing maintenance effort through software operation knowledge: An eclectic empirical evaluation. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on Software Maintenance and Reengineering*, pages 201–210. IEEE, 2011. 22

[29] S.C.B. de Souza et al. A Study of the Documentation Essential to Software Maintenance. *Proceedings of the 23rd annual international conference on Design of communication documenting & designing for pervasive information - SIGDOC '05*, page 68, 2005. 1

[30] M. Sullivan and R. Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, volume 21, pages 2–9, 1991. 9

[31] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* Wiley Publishing, 2009. 22

[32] W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, volume 136. 2011. ix, 20

[33] H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. Prom 6: The process mining toolkit. *Proceedings of BPM Demonstration Track*, 615:34–39, 2010. 21

[34] H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. Xes, xesame, and prom 6. In *Forum at the Conference on Advanced Information Systems Engineering (CAiSE)*, pages 60–75. Springer, 2010. ix, 21, 25

[35] J.M.E.M. van der Werf and S. Brinkkemper. Incorporating Big Data in the Architecture Process. ix, 22, 23

[36] J.M.E.M. van der Werf and H.M.W. Verbeek. Online compliance monitoring of service landscapes. In *International Conference on Business Process Management*, pages 89–95. Springer, 2014. 23

[37] R. Wettel and M. Lanza. CodeCity: 3D Visualization of Large-Scale Software. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, page 921, New York, New York, USA, 2008. ACM Press. 15

[38] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 551, New York, New York, USA, 2011. ACM Press. 15

[39] E.J. Weyuker. Evaluation techniques for improving the quality of very large software systems in a cost-effective way. *Journal of Systems and Software*, 47(2):97–103, 1999. 1

[40] D. Winkler, S. Biffl, and J. Bergsmann. Software quality: The future of systems- and software development: 8th International Conference, SWQD 2016 Vienna, Austria, January 1821, 2016 Proceedings. In *Lecture Notes in Business Information Processing*, volume 238, 2016. 13

Appendix A

Figures and Tables

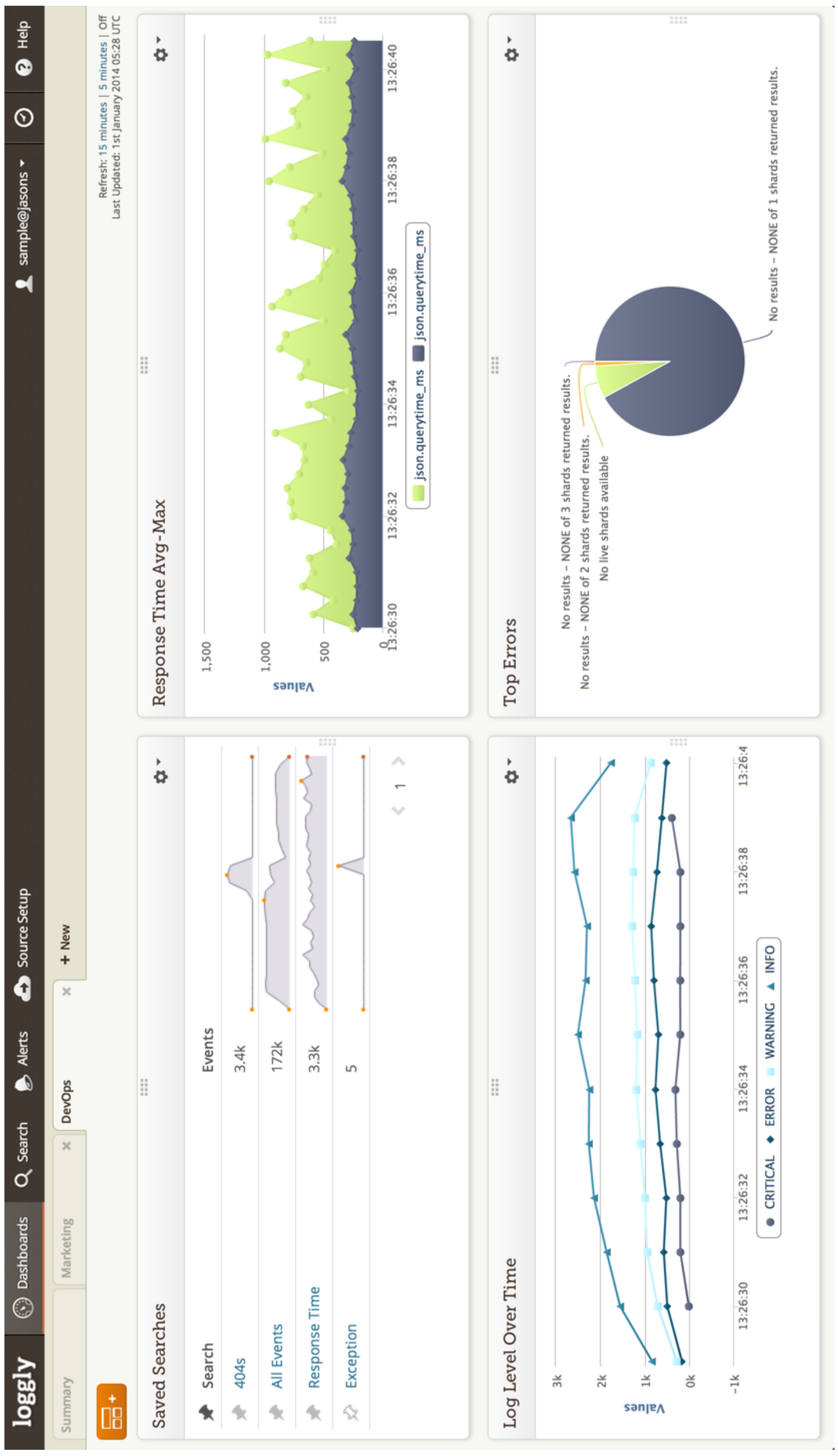


Figure A.1: The Loggly dashboard as found on their website[4]

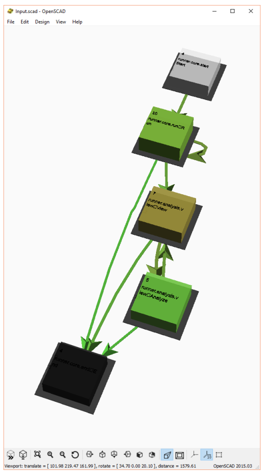


Figure A.2: ArchitectureCity with building colors representing the fan in/out ratio as gradient and opacity as absolute fan in/out values

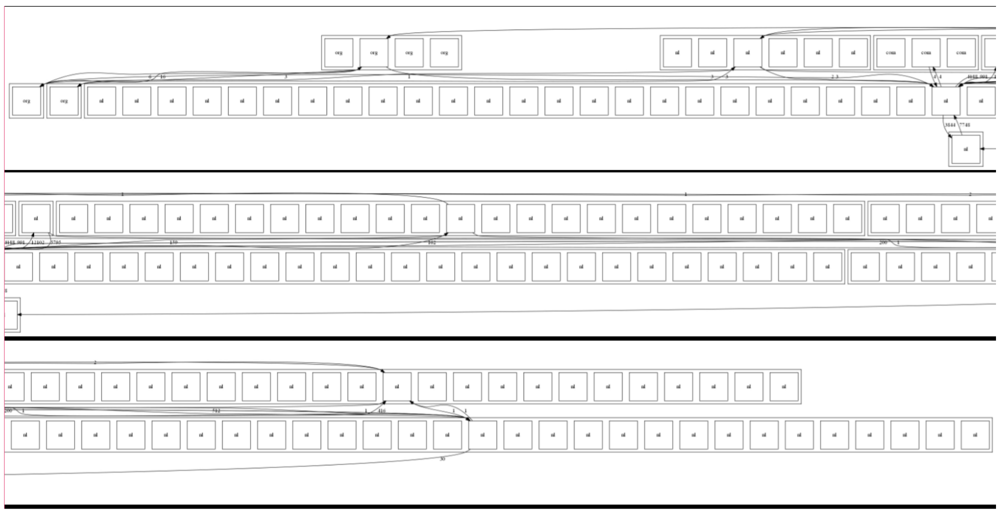


Figure A.3: Graphviz using the built in sub-graphs rendered using the dot placement algorithm, split into three parts to better fit on the page

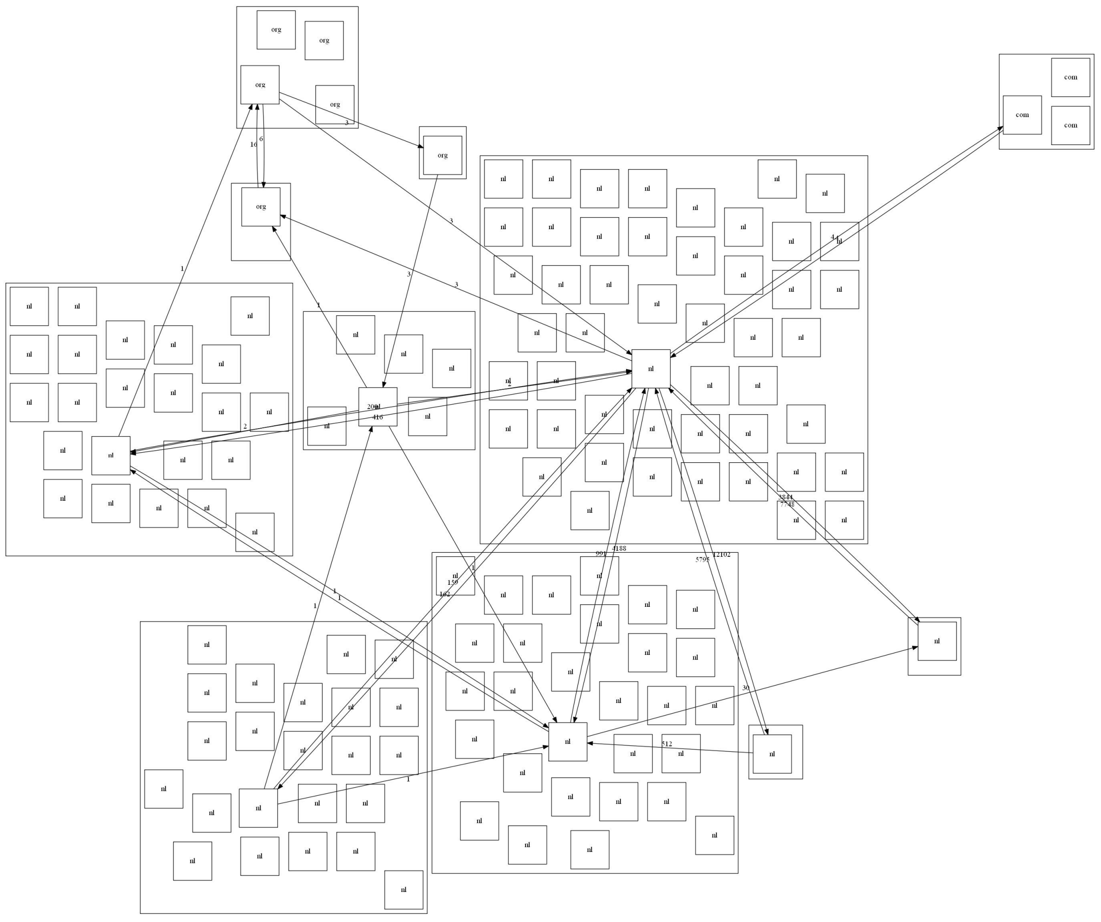


Figure A.4: Graphviz using the built in sub-graphs rendered using the fdp placement algorithm

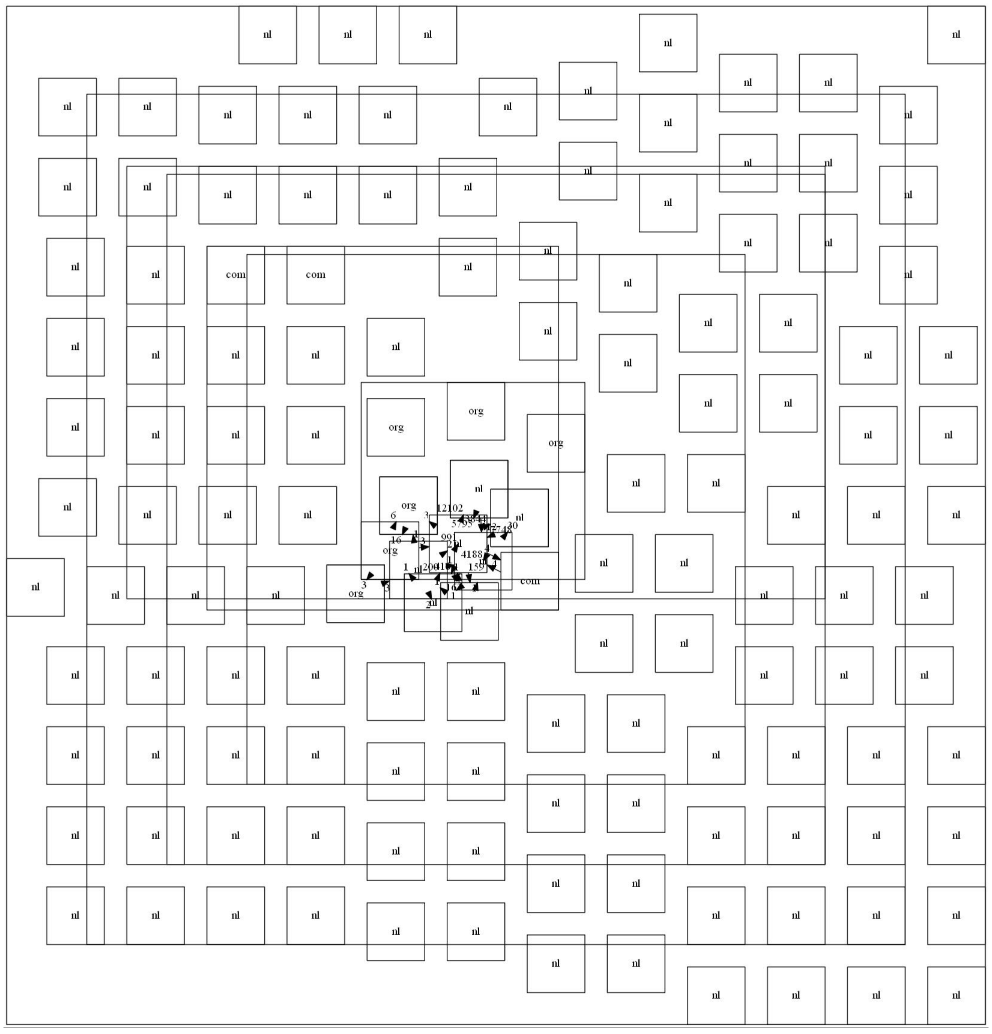


Figure A.5: Graphviz using the built in sub-graphs rendered using the neato placement algorithm

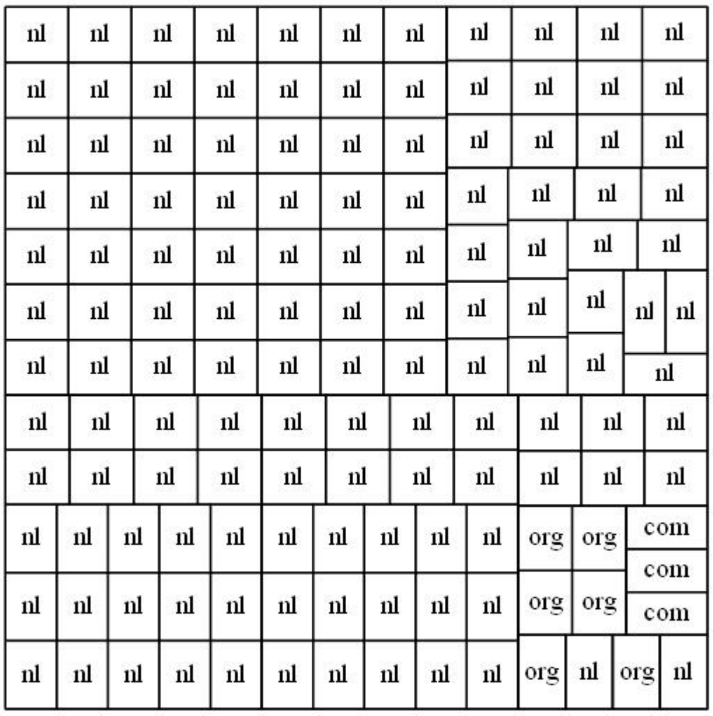

nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl nl
nl nl nl nl nl nl nl nl nl nl org org com
com
nl nl nl nl nl nl nl nl nl nl org org com
nl nl nl nl nl nl nl nl nl nl org nl org nl

Figure A.6: Graphviz using the built in sub-graphs rendered using the patchwork placement algorithm

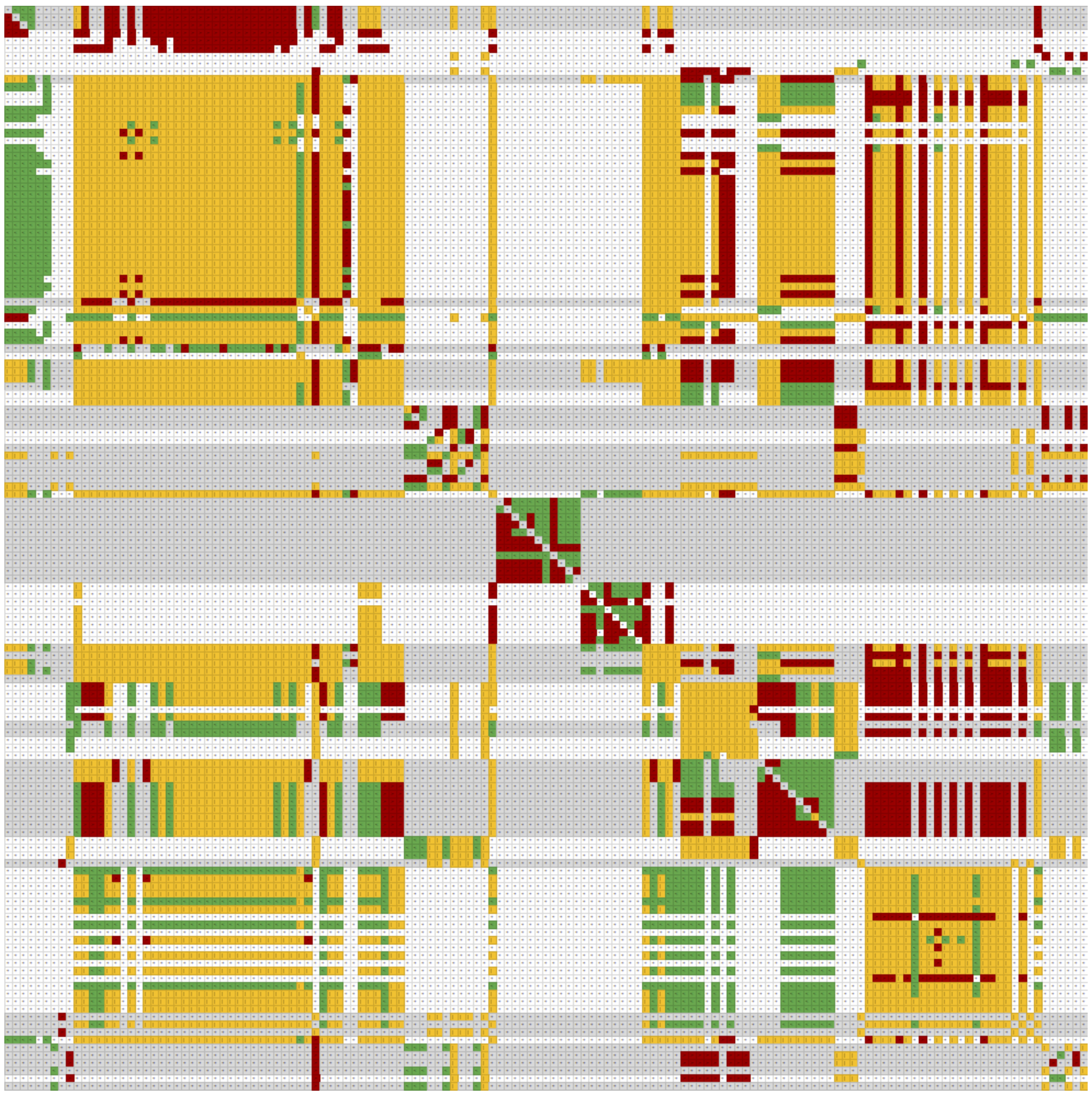


Figure A.7: A matrix overview for k = infinity, split time = 30 minutes, of the DL case log of one system over a three day time interval. Actions are sorted by package and then alphabetical. Packages are horizontally indicated using the gray-white alternating background, the same order is present on the vertical axis.

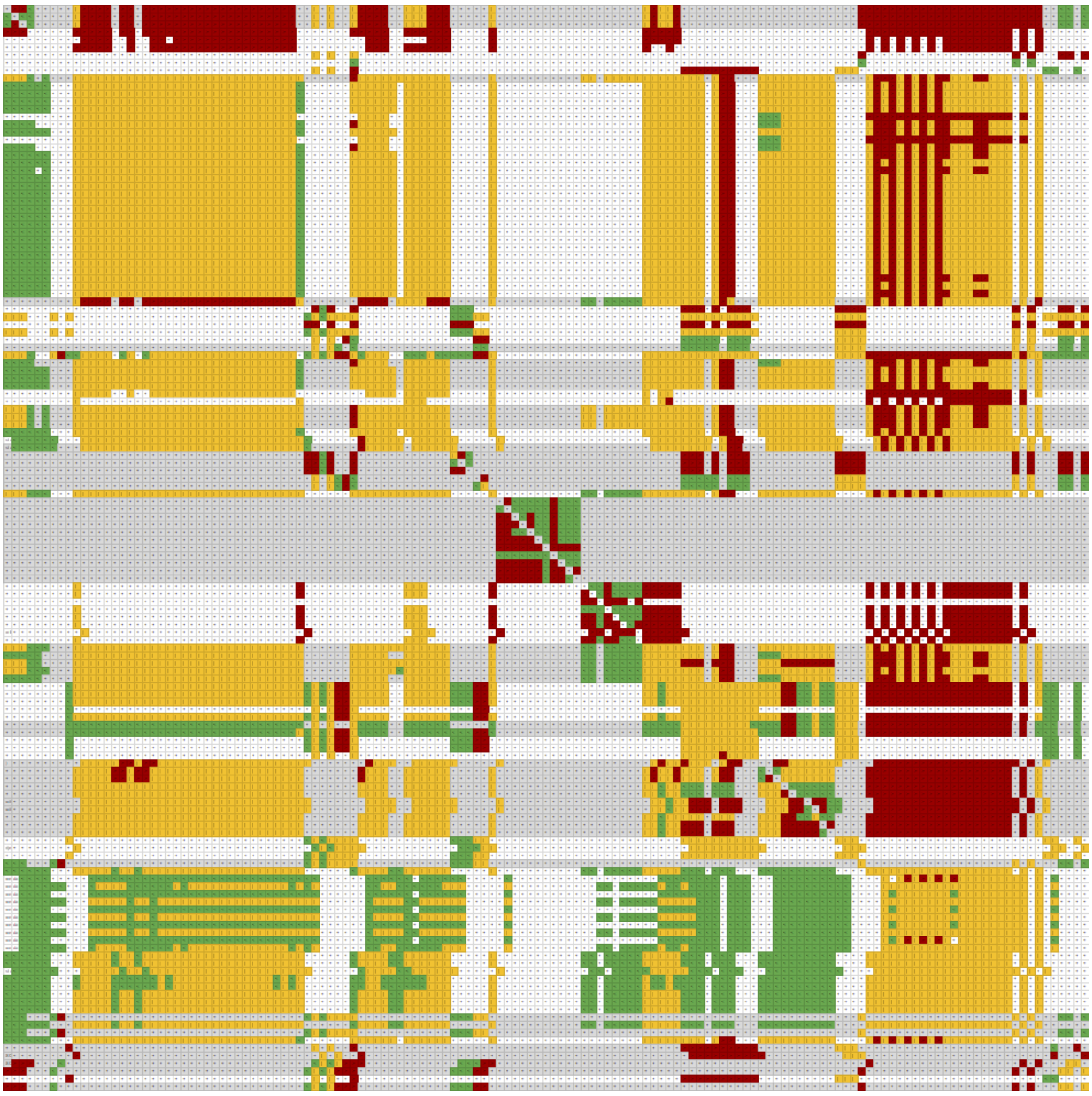


Figure A.8: A matrix overview for k = infinity, split time = 22 hours, of the DL case log of one system over a three day time interval. Actions are sorted by package and then alphabetical. Packages are horizontally indicated using the gray-white alternating background, the same order is present on the vertical axis.

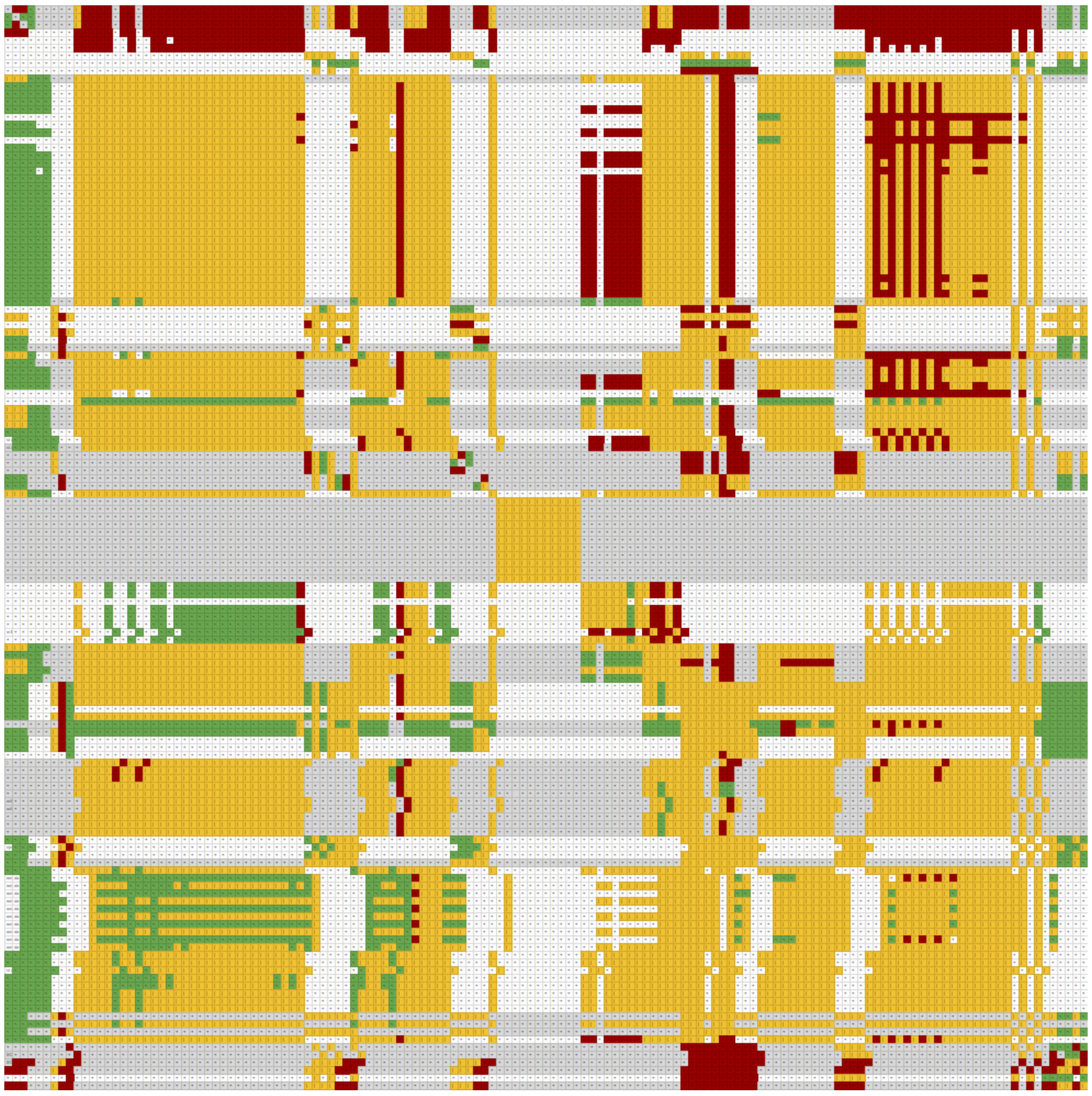


Figure A.9: A matrix overview for k = infinity, no split time, of the DL case log of one system over a three day time interval. Actions are sorted by package and then alphabetical. Packages are horizontally indicated using the gray-white alternating background, the same order is present on the vertical axis.

.

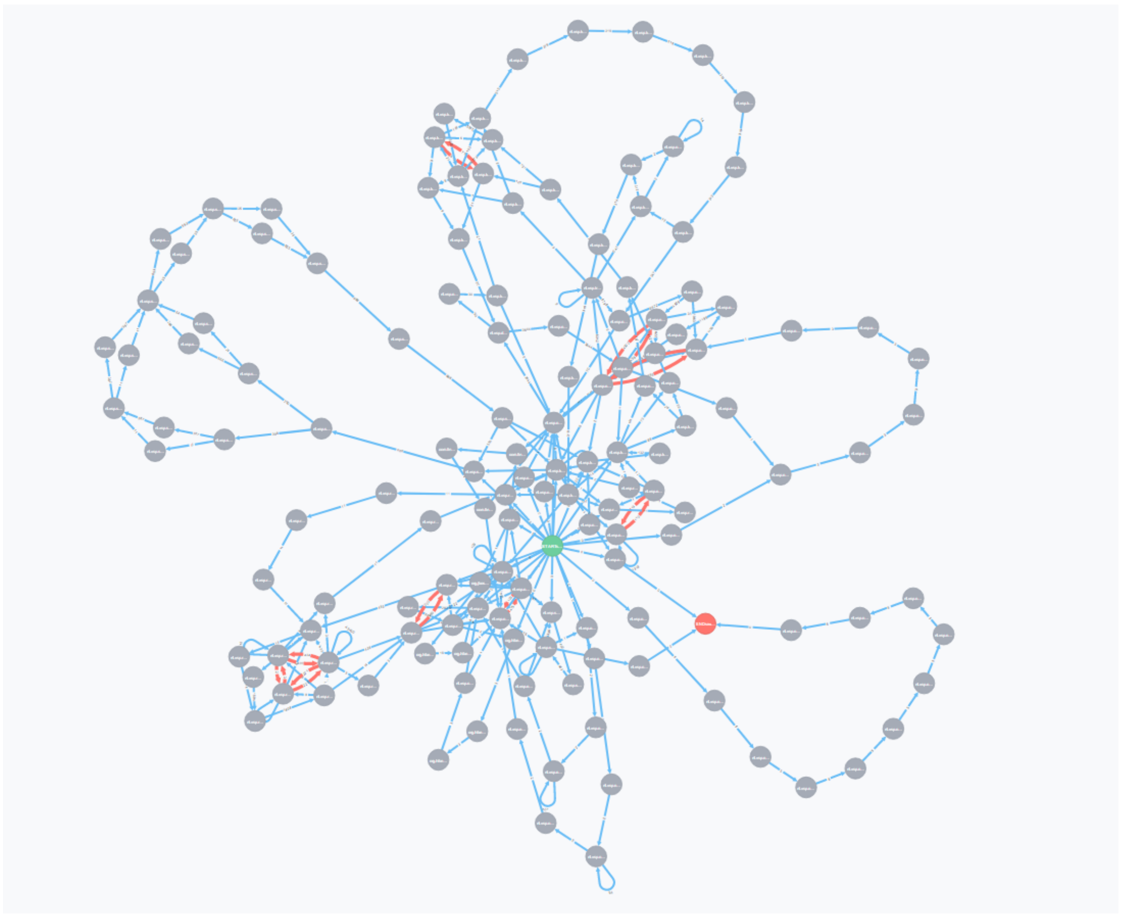
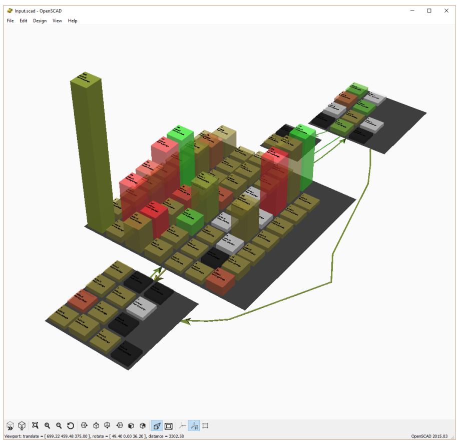
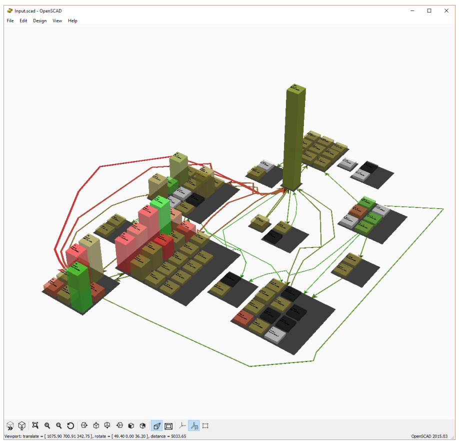
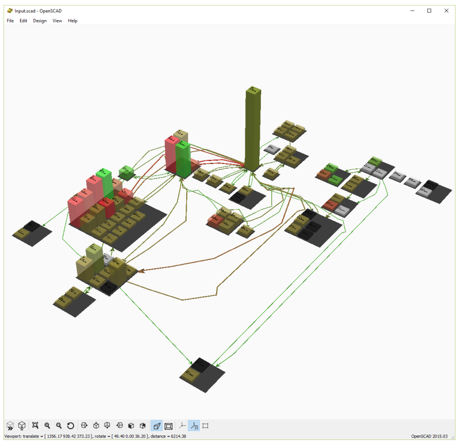
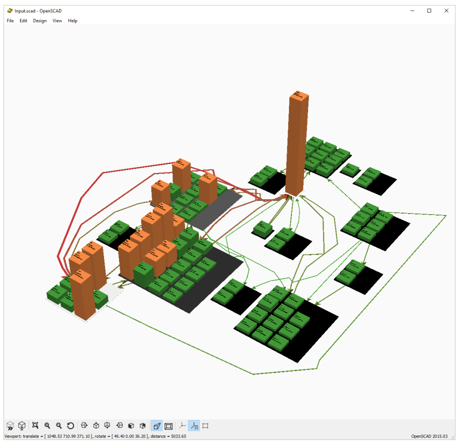


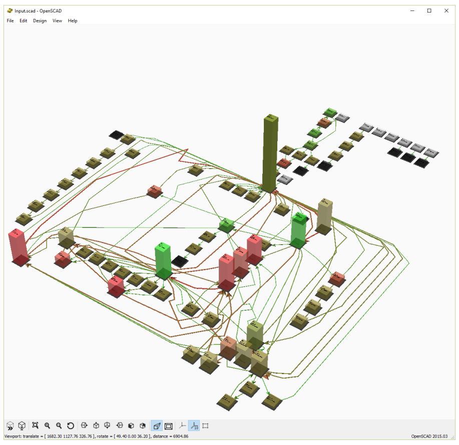
Figure A.10: A workflow net of the back-end-1 process of DL


(a) Package clustering with depth of 1 with fan coloring


(b) Package clustering with depth of 3 with fan coloring


(c) Package clustering with depth of 5 with fan coloring


(d) Package clustering with depth of 3 with call-count coloring


(e) Without clustering with fan coloring

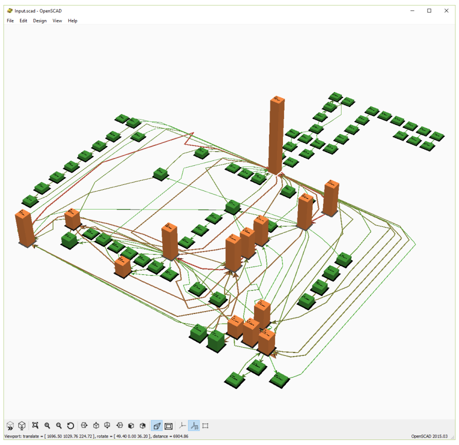
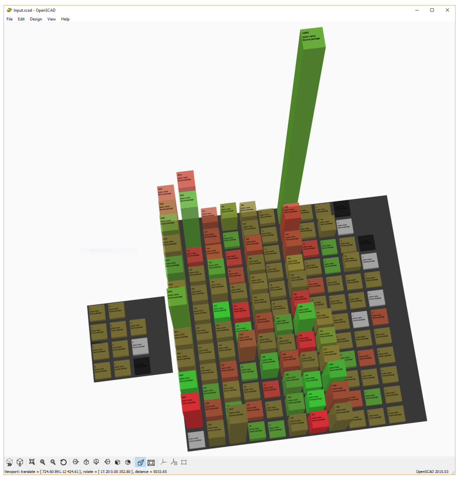
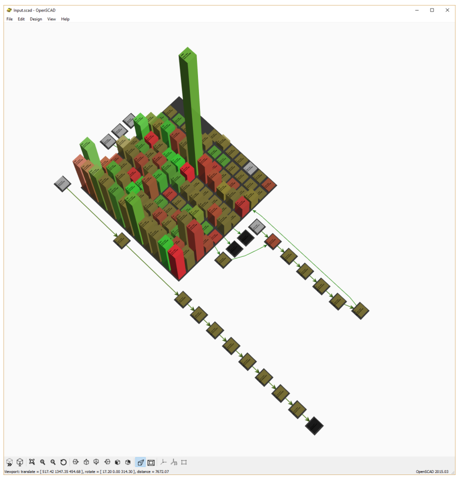

(f) Without clustering with call-count coloring

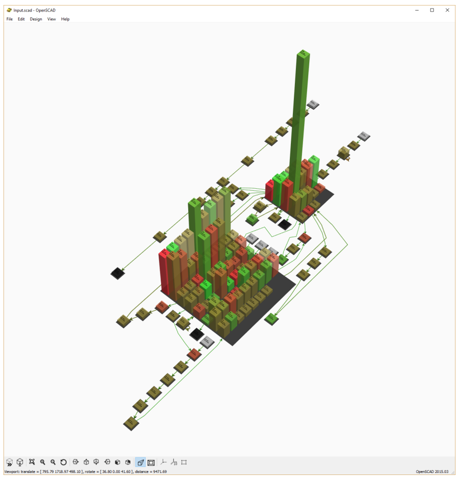
Figure A.11: A visualization of ArchitectureCity for varying levels of package clustering and coloring methods. Logs are from front-c-1 with a split time of 30 minutes.



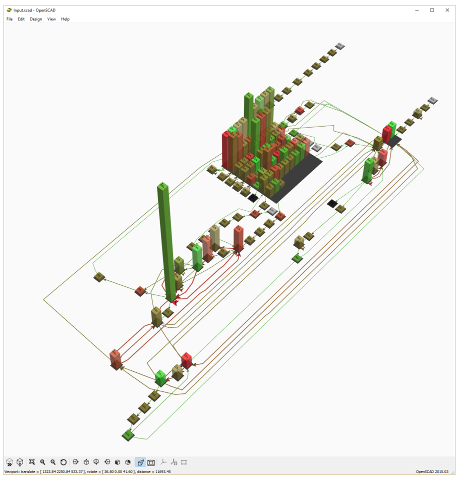
(a) Fan in/out clustering using cluster count of 2 with fan coloring



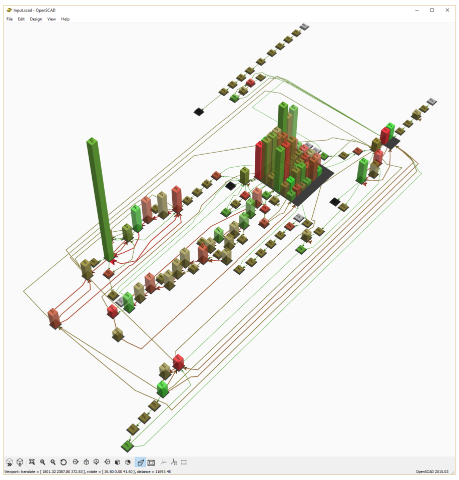
(b) Fan in/out clustering using cluster count of 25 with fan coloring



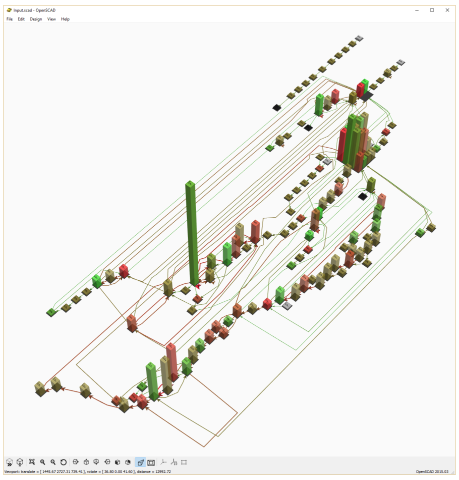
(c) Fan in/out clustering using cluster count of 50 with fan coloring



(d) Fan in/out clustering using cluster count of 75 with fan coloring

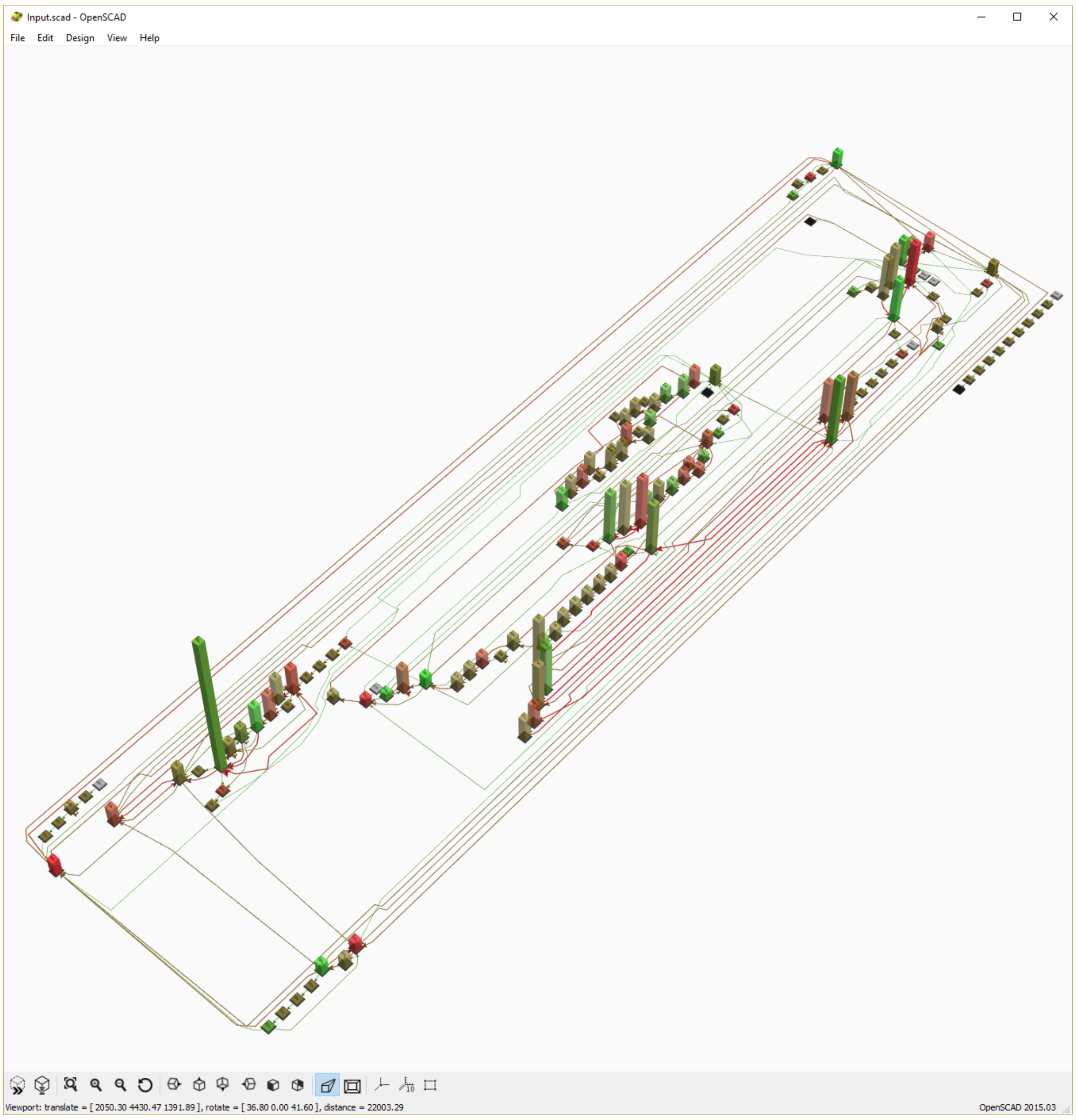


(e) Fan in/out clustering using cluster count of 100 with fan coloring



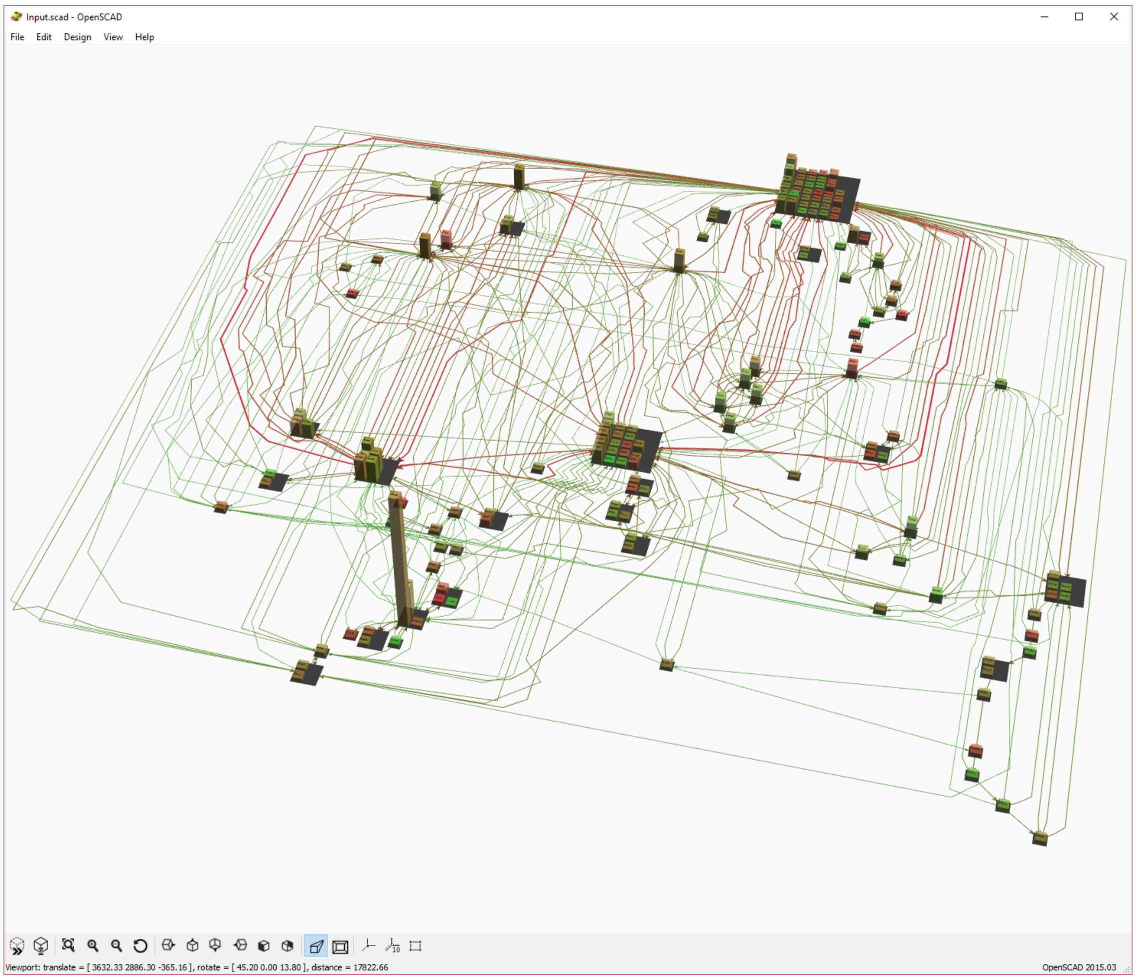
(f) Fan in/out clustering using cluster count of 125 with fan coloring

Figure A.12: A visualization of ArchitectureCity of the DL system for varying levels of fan in/out clustering. Logs are from front-c-1 with a split time of 30 minutes.



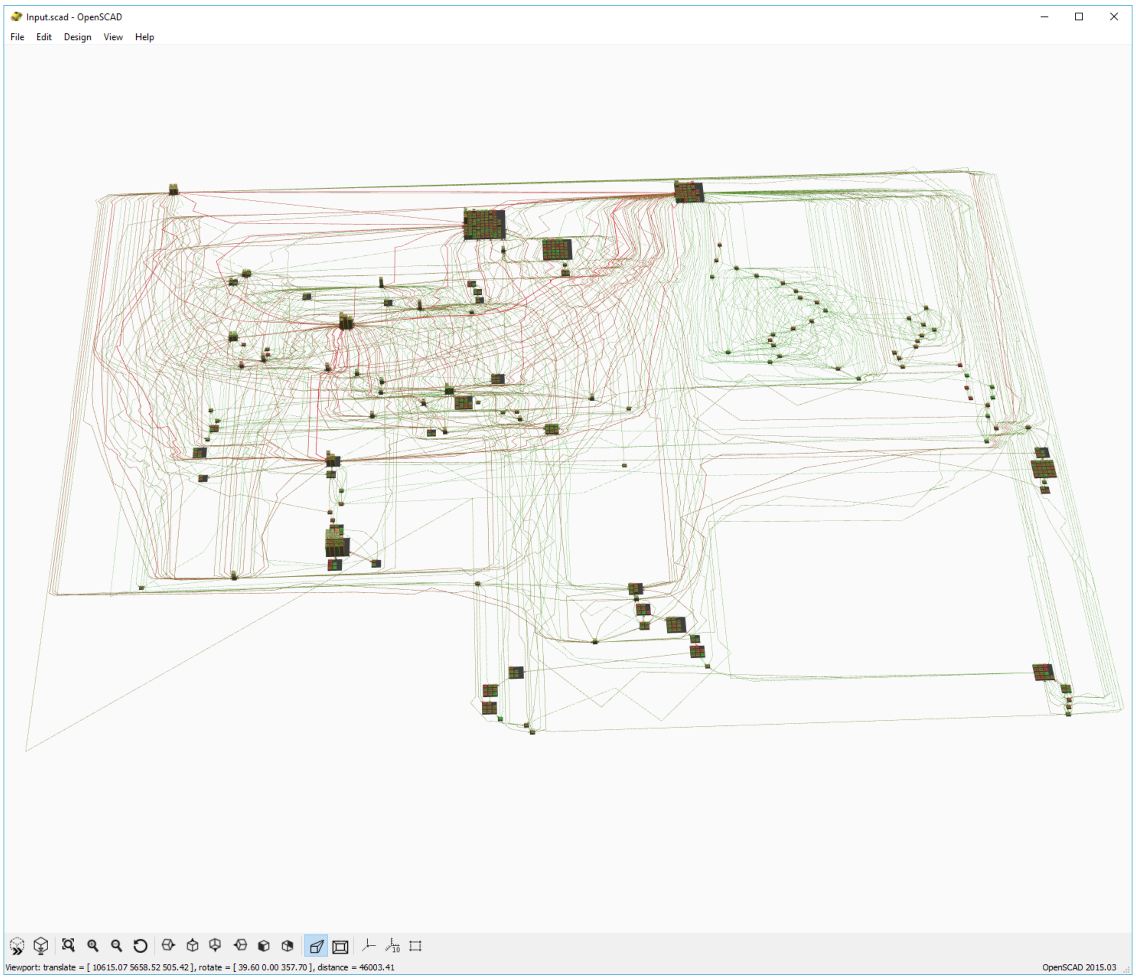
```
Viewport: translate = [ 2050.30 4430.47 1391.89 ], rotate = [ 36.80 0.00 41.60 ], distance = 22003.29
```

Figure A.13: A visualization of ArchitectureCity of the DL system using fan in/out clustering with an infinite cluster count. Logs are from front-c-1 with a split time of 30 minutes.



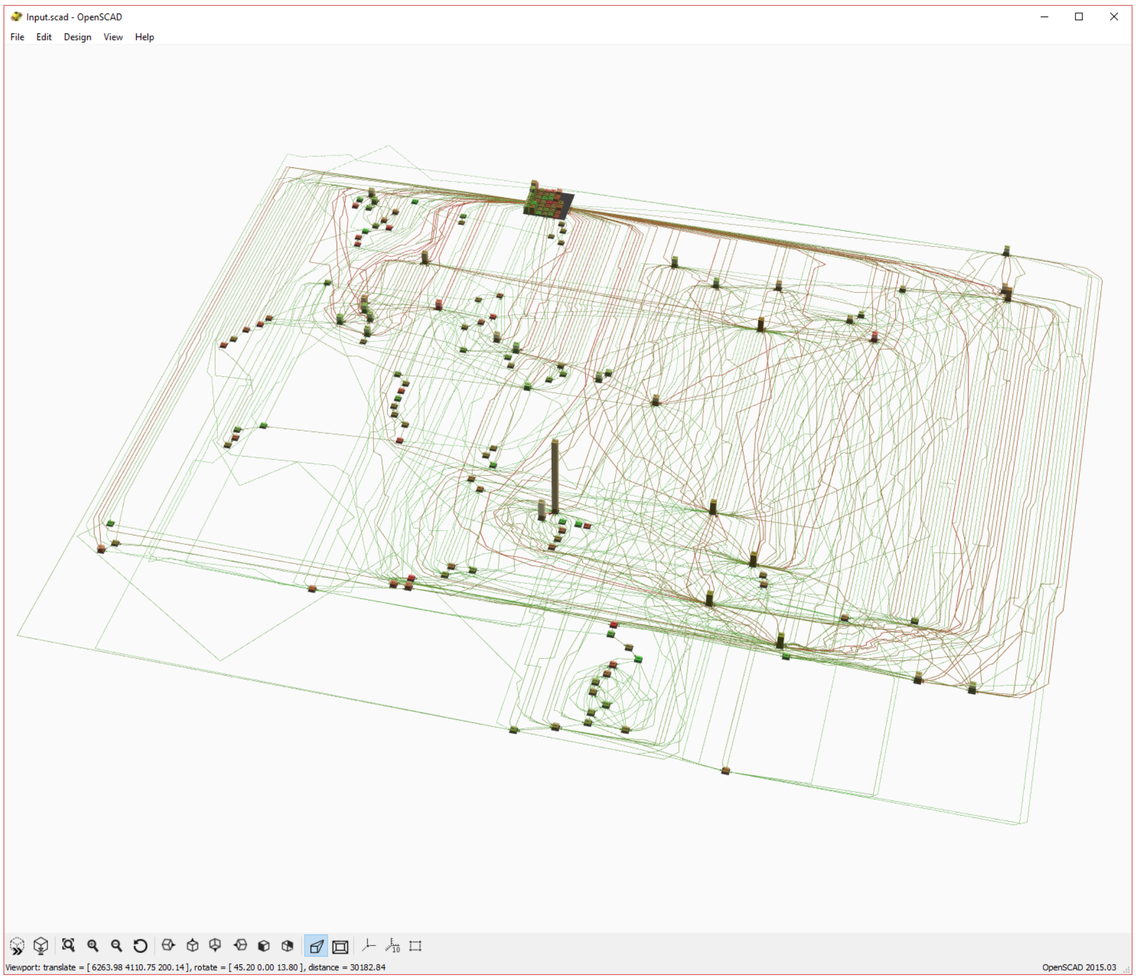
Viewport: translate = [3632.33 2886.30 -365.16], rotate = [45.20 0.00 13.80], distance = 17822.66

Figure A.14: A visualization of ArchitectureCity of the FI system with package level 2 with pruning.



```
Viewport: translate = [ 10615.07 5658.52 505.42 ], rotate = [ 39.60 0.00 357.70 ], distance = 46003.41
```

Figure A.15: A visualization of ArchitectureCity of the FI system with package level 2 without pruning.



Viewport: translate = [6263.98 4110.75 200.14], rotate = [45.20 0.00 13.80], distance = 30182.84

OpenSCAD 2015.03

Figure A.16: A visualization of ArchitectureCity of the FI system with package level 3 with pruning.

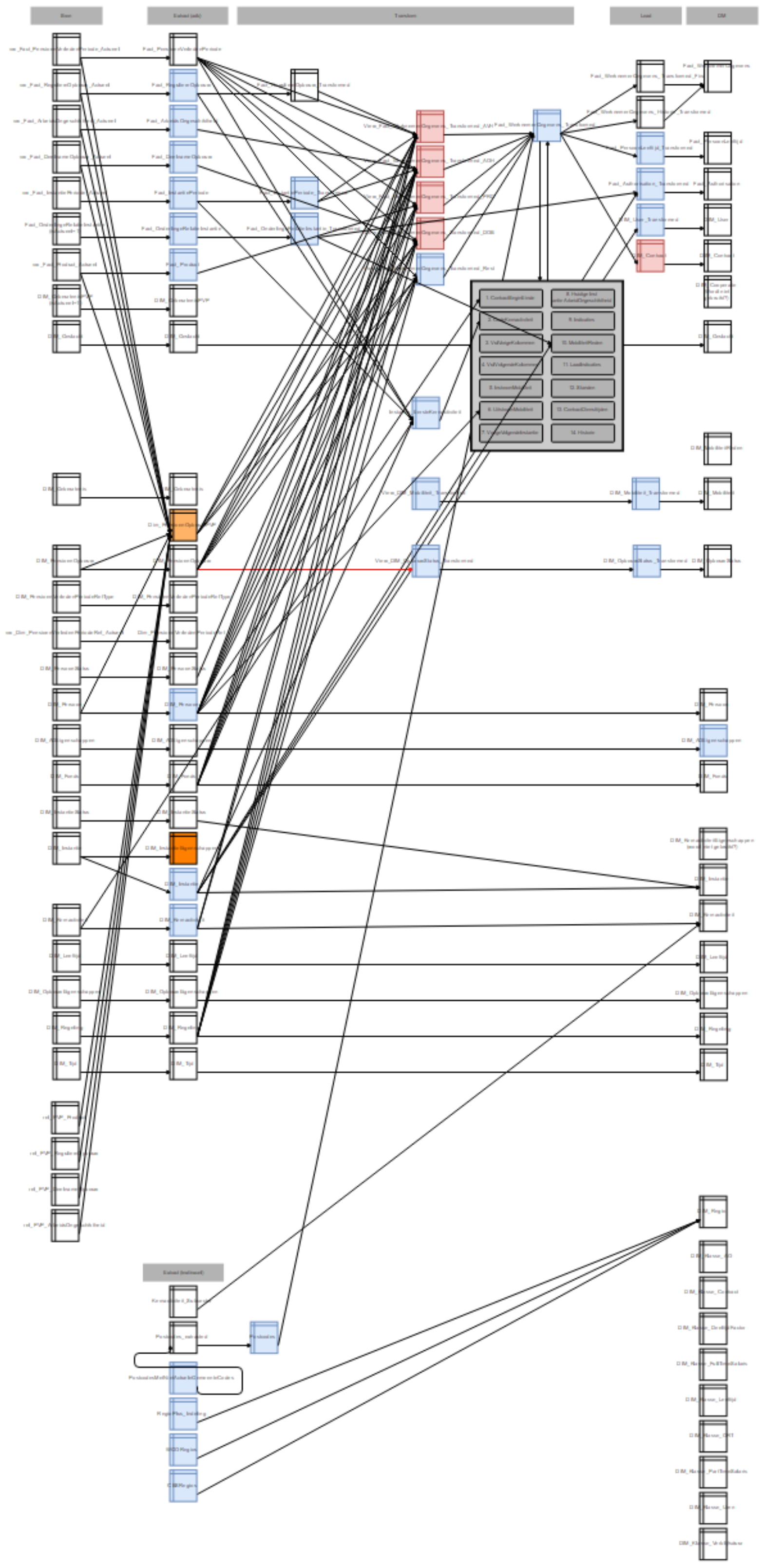


Figure A.17: An overview of the information flow of the old FI system with a lot of technical debt.

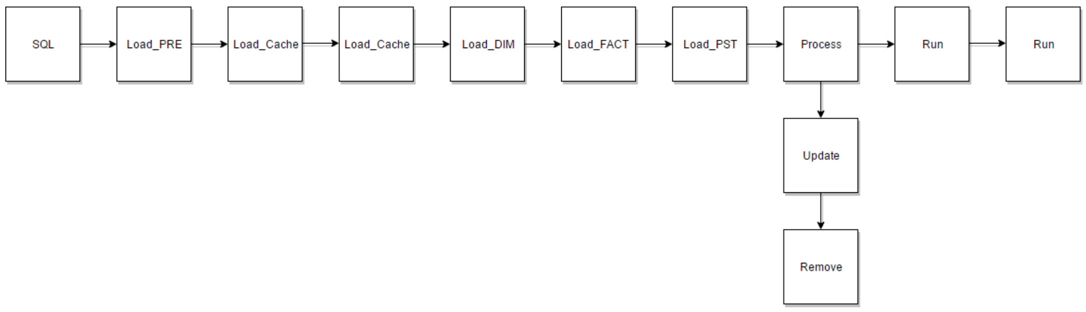


Figure A.18: An overview of the information flow of the new FI system.

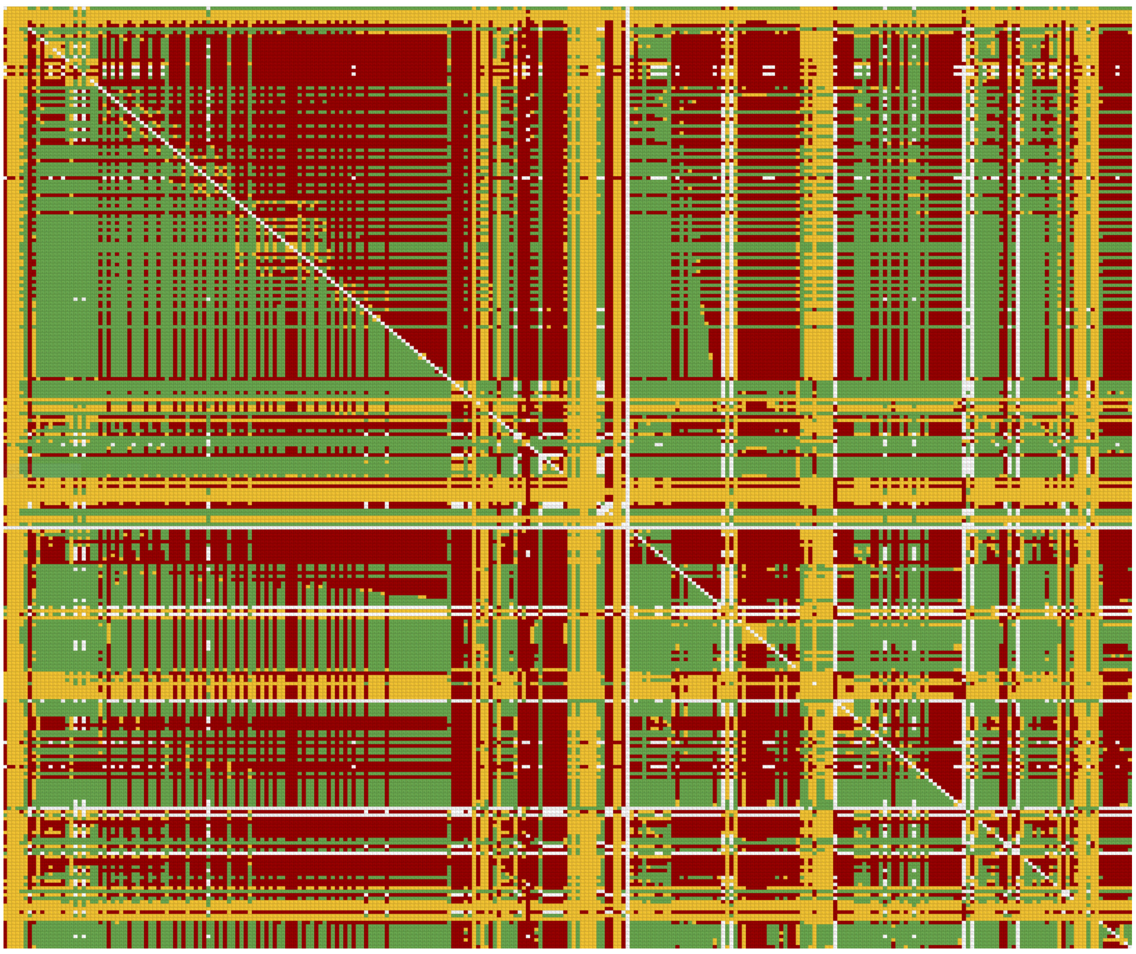


Figure A.19: A Matrix overview of the old FI system.

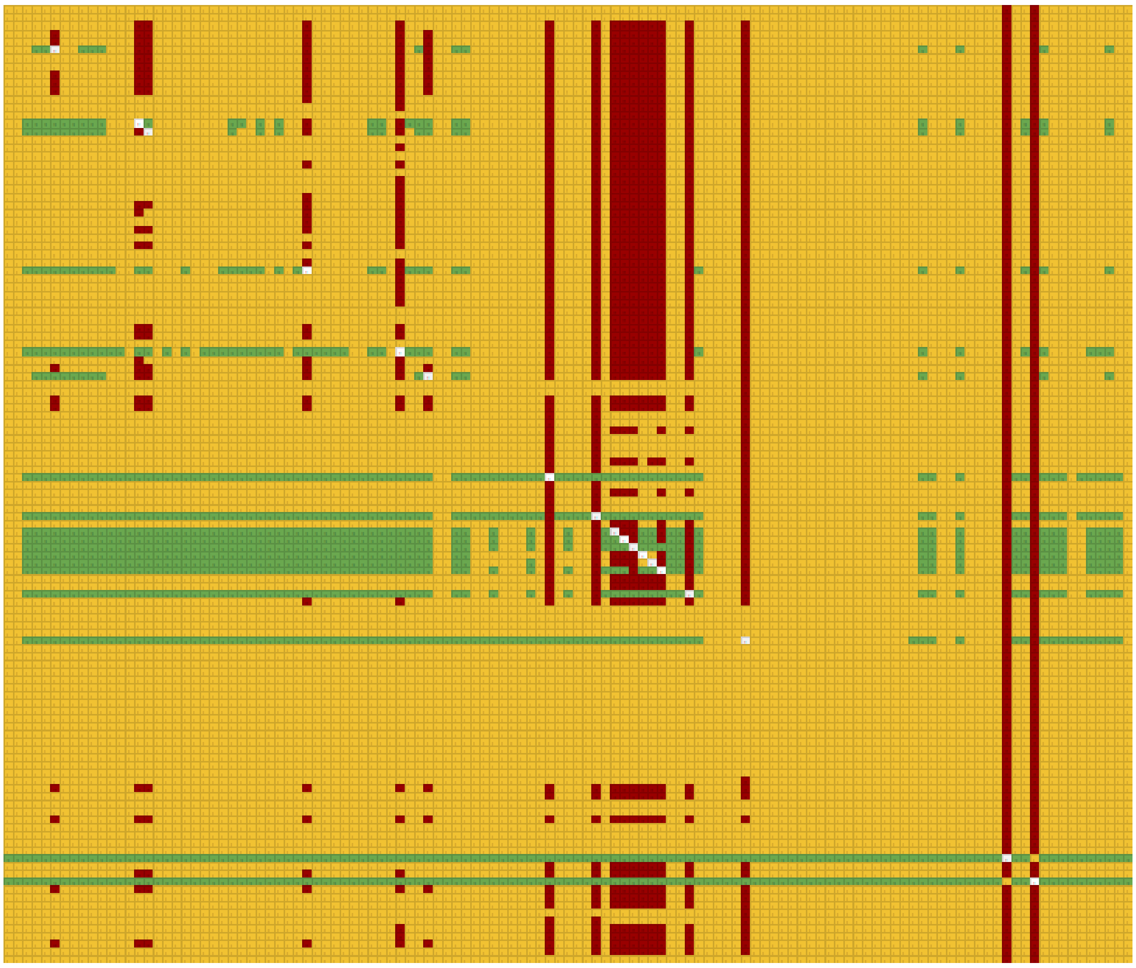
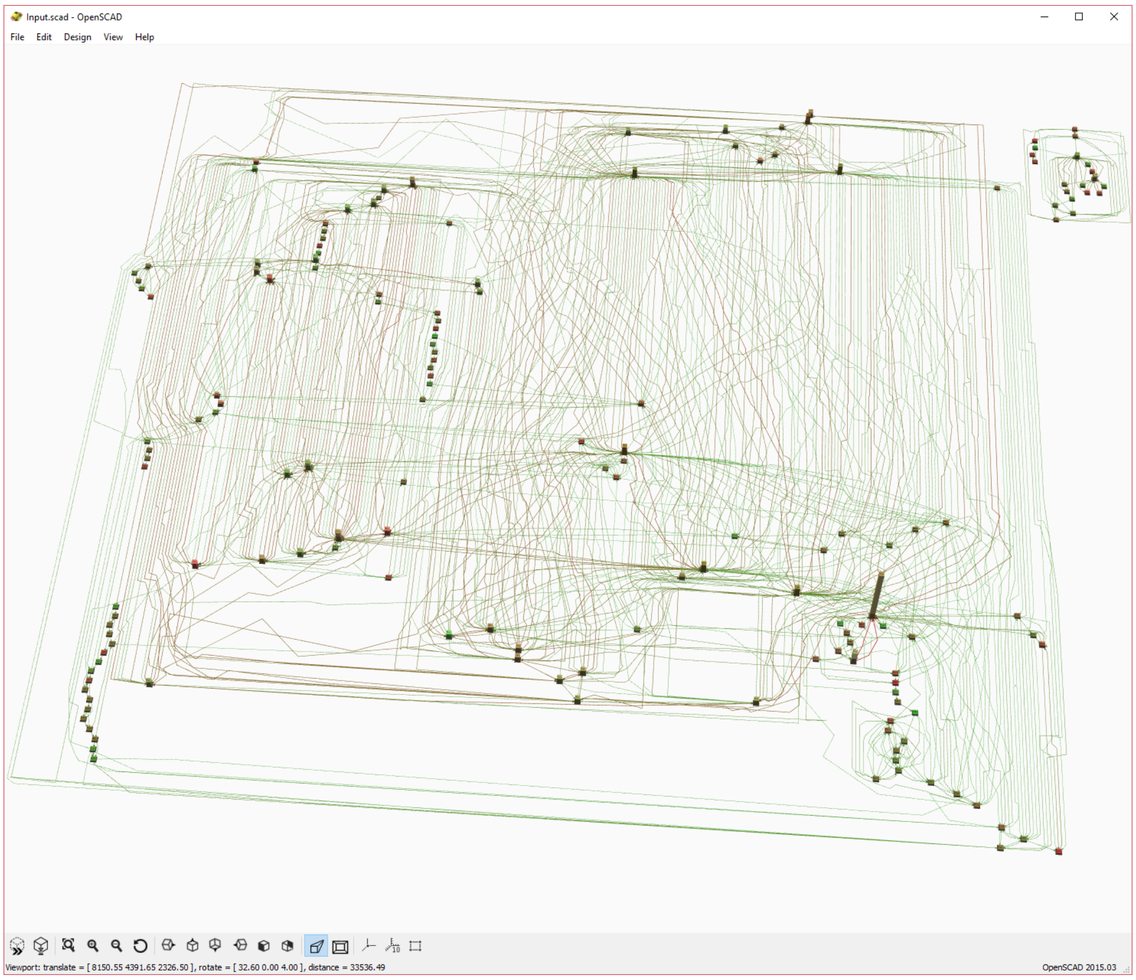
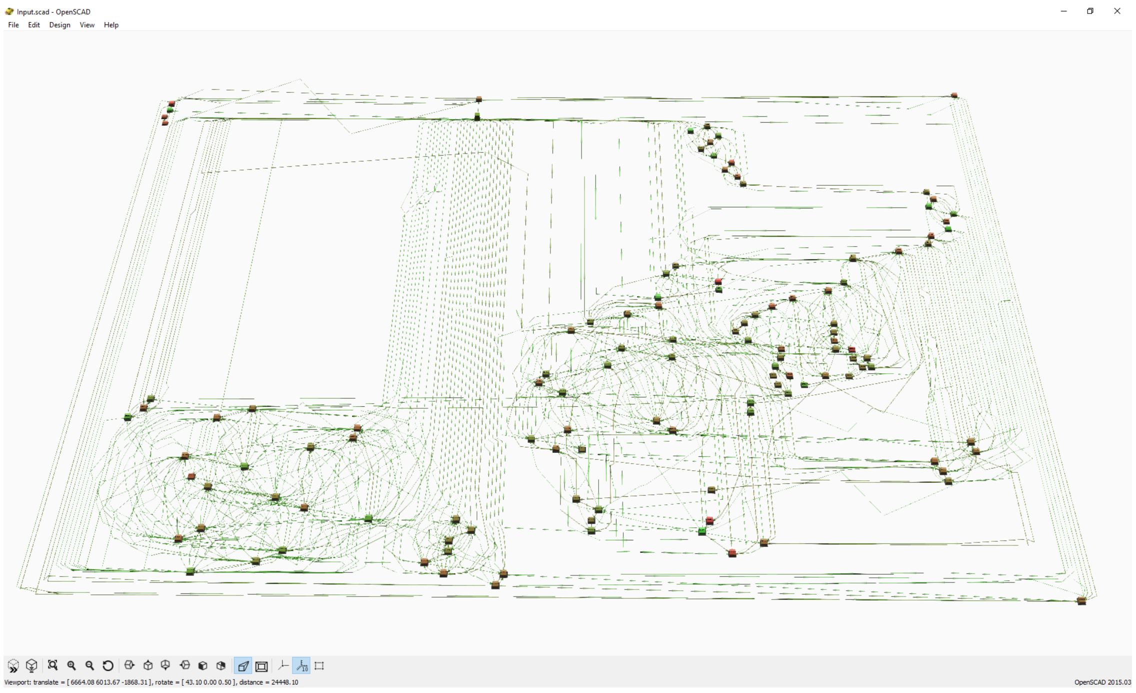


Figure A.20: A Matrix overview of the new FI system.



```
Viewport: translate = [ 8150.55 4391.65 2326.50 ], rotate = [ 32.60 0.00 4.00 ], distance = 33536.49
```

Figure A.21: An overview of the architecture of the FI systems with fan in/out clustering and no max cluster count.



Viewport: translate = [6664.08 6013.67 -1868.31], rotate = [43.10 0.00 0.50], distance = 24448.10

Figure A.22: An overview of the architecture of the new FI system without any clustering or pruning.

Appendix B

Listings

```
datetime,              package,                  processID,      actionID
15-09-2016 03:30:02,00 [runner.core.start]       (7)             Start
15-09-2016 04:36:05,00 [runner.core.run]         (7)             Run
15-09-2016 04:38:07,00 [runner.core.end]         (7)             End
15-10-2016 02:36:15,00 [runner.core.start]       (29)            Start
15-10-2016 02:42:05,00 [runner.core.run]         (29)            Run
15-10-2016 02:48:55,00 [runner.core.run]         (29)            Run
15-10-2016 03:22:25,00 [runner.core.run]         (29)            Run
15-10-2016 03:48:52,00 [runner.core.run]         (29)            Run
15-10-2016 04:30:01,00 [runner.core.run]         (29)            Run
15-10-2016 04:45:11,00 [runner.core.run]         (29)            Run
15-10-2016 04:57:05,00 [runner.core.run]         (29)            Run
15-10-2016 05:48:05,00 [runner.analysis.view]    (29)            View
15-10-2016 05:49:05,00 [runner.analysis.view]    (29)            Analyze
15-10-2016 05:54:05,00 [runner.analysis.view]    (29)            View
15-10-2016 05:54:45,00 [runner.core.end]         (29)            End
15-10-2016 14:00:00,00 [runner.core.start]       (17)            Start
15-10-2016 14:01:05,00 [runner.core.run]         (17)            Run
15-10-2016 14:11:00,00 [runner.analysis.view]    (17)            View
15-10-2016 14:12:00,00 [runner.analysis.view]    (17)            Analyze
15-10-2016 14:42:30,00 [runner.analysis.view]    (17)            View
15-10-2016 15:30:05,00 [runner.analysis.view]    (17)            Analyze
15-10-2016 15:30:59,00 [runner.core.end]         (17)            End
15-11-2016 03:36:05,00 [runner.core.start]        (2)            Start
15-11-2016 03:42:05,00 [runner.core.run]          (2)            Run
15-11-2016 03:50:05,00 [runner.analysis.view]     (2)            View
15-11-2016 03:51:05,00 [runner.analysis.view]     (2)            Analyze
15-11-2016 03:56:05,00 [runner.analysis.view]     (2)            View
15-11-2016 03:57:05,00 [runner.analysis.view]     (2)            Analyze
15-11-2016 03:58:05,00 [runner.analysis.view]     (2)            View
15-11-2016 03:59:05,00 [runner.core.end]          (2)            End
```

Listing B.1: The running app event log sorted by datetime

```
module Building(x,y,z,s,c, callCount = 0, label="", secondlbl=""){
      textSize = 6;
        translate([x,y,0])
        {
          // Cube
          color(c){
            if(s < 96)
            translate([0,0,-1])
              cube([s,s,z]);
            else
              cube([s,s,z]);
          }
          // Text
          color([0,0,0,1]){
            if(secondlbl == ""){
              translate([2, s/2, z])
              text(label, size = textSize);
            }
            else{
              translate([2, s/2 + 5, z])
              text(label, size = textSize);
            }

            translate([2, s/2 - textSize, z])
            text(secondlbl, size = textSize);

            translate([4, s/2 + 20, z])
            text(callCount, size = textSize);
          }
      }
}

module Road(point1, point2, size, roadColor = [0.5,0.5,0.5,1]){
    color(roadColor)
    hull()
    {
      translate(point1) circle(size);
      translate(point2) circle(size);
    }
}

module Arrow(point, angle, arrowColor, size=5){
  color(arrowColor)
  translate(point)
  rotate(angle - 90)
  translate([-size * 2, -size * 2, -size * 0.5])
  polyhedron
    (points = [
        [0,0,0],[2 * size, 1 * size, 0],[4 * size, 0,0],
        [2 * size, 5 * size, 0],[2 * size, 2 * size, size]
      ],
     faces = [[0,1,4],[0,1,3],[0,4,3],[2,1,4],[2,1,3],[2,4,3]]
     );
}
```

Listing B.2: The code for running the build command created by the ArchitectureCity visualizer

ArchitectureCity: Visualizing The Dynamics of Software through Software Operation Data

Rens M. Rooimans†, Jan Martijn E.M. van der Werf†, Jurriaan Hage†, Frans van den Berg‡

† Utrecht University ‡ Omnext B.V.

{r.m.rooimans, j.m.e.m.vanderwerf, j.hage}@uu.nl

frans.van.den.berg@omnext.net

ABSTRACT

Ideally, software documentation follows the actual implementation. However, due to a plethora of reasons, many software systems have outdated or incomplete architecture documentation. In this paper, we present an approach that relies on the actual operation of software to gain new insights for software architects. Based on the software operation data generated by the system, we employ architecture mining to extract and enhance operational data to support the software architect. For this, we have developed the Architectural Intelligence Mining Framework, and more specifically, ArchitectureCity, which uses the analogy of cities to visualize the runtime of software: buildings, representing individual architectural elements are grouped in districts based on different clustering techniques, and streets depict the traffic between the different districts. We have applied the framework to a real life case study. The visualisation techniques were positively received, which shows the potential of the proposed techniques.

1. INTRODUCTION

Organizations nowadays rely more and more on very large software systems that are in place for a large number of years [24]. As a consequence, these systems tend to be irreplaceable within the organization. Over the years, maintenance of such systems becomes a burden to the organization, e.g., because used technology is phased out, or knowledgeable engineers leave the organization. Due to the many changes and additions over the years, software erosion [2] lurks: documentation is outdated and does not fully match the implemented system [7]. This is especially true for the software architecture of such systems.

Ideally, the software architecture, i.e., the set of structures needed to reason about the system, which comprises software elements, relations among them and properties of both [2], reflects the software system accurately. Unfortunately, architectural and technical documentation are, in many cases, far behind on the source code comments and experience levels. The latter are also the most volatile, as comments and source code change constantly, and experienced staff can leave at any moment. Fields like Software Architecture Compliance Checking (SACC) [9] and Software Architecture Reconstruction (SAR) [5] provide support to close the gap between architectural documentation and the actually produced software artifacts.

Many of the current SACC and SAR tools rely on static information, i.e., the source code and other artifacts created at design time. For example, HUSACCT [13] builds upon the dependencies induced by the source code. CodeCity [23] takes the software units as input, and visualizes the source code in the form of a city: each unit becomes a building with a certain size and colour, depending on the size of the unit, and the chosen metric.

In this paper, we follow a different approach by analyzing the runtime of actual software systems to provide insights and architectural documentation relevant for the architects. Runtime analysis in itself is not new, e.g. for feature location [8], and reflexion modeling [3]. Software systems typically log events raised by the system [10, 22, 25], like the start or completion of a functional element in the system. Logs that describe the operation of the software system is what we refer to as software operation data [17, 20].

Analyzing software operation data has the potential to reveal many relevant insights to the software architect, e.g. the discovery of rarely used parts of the system, bottlenecks or undesired and unintended dependencies [17, 21]. To aid the architect in analyzing the software operation data, we propose a new visualization technique: ArchitectureCity. Architectural elements become districts or buildings, depending on their hierarchy, the size and metrics of the different elements relate to their appearance in the software operation data: a high building corresponds to a frequently used element. In addition, the size and colour of the streets represent the different calls between the elements.

The remainder of this paper is structured as follows. Section 2 introduces architecture mining and the required elements in the software operation data. Next, we present ArchitectureCity and the ideas behind the used visualization techniques (Section 3), and an initial evaluation at a case study organization (Section 4). Section 5 concludes the paper.

2. ARCHITECTURE MINING

This paper focuses on the use of runtime software opera-

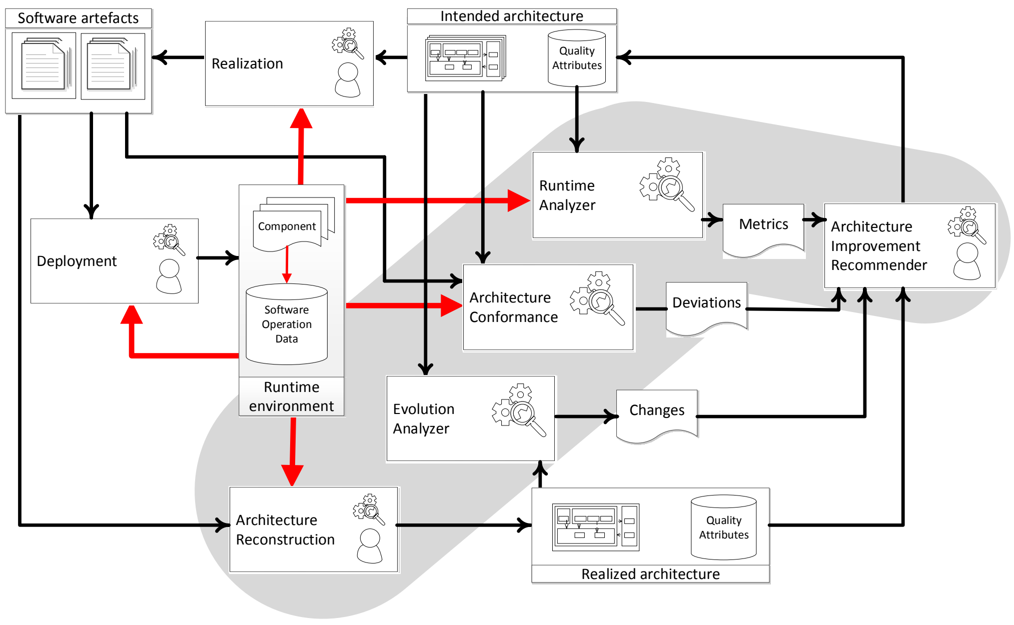


Figure 1: Architectural Intelligence Framework based on software operation data [20].

tion data to provide new insights to the software architect. Its analysis has the potential to close the loop between software architecture and the realized software system by monitoring and analyzing the actual realized system. In this way, data analytics techniques like Process Mining [16] can provide new insights to the architect.

Architecture mining is defined as the collection, analysis, and interpretation of software operation data to foster architecture evaluation and evolution [20]. The five main activities identified in architecture mining, depicted in Figure 1, include architecture reconstruction, architecture conformance, runtime analysis, evolution analysis and architecture improvement recommendation. The arrows indicate the flow of information between the activities, whereas the red arrows indicate which steps can benefit from software operation data. Notice that the arrows do not indicate any order between the activities.

2.1 Software Operation Data

One way to monitor the system operation is by gathering and storing these events [16,21]. Each event is generated by some resource and a time window in which it occurred [19]. Events can contain additional data, such as the function being called, its parameters, etc. In this way, each run of the system results in a trace of consecutive events. An excerpt of a trace is shown in Table 1. In this example, 8 events occurred on 3 different processes, with different actions. Typically, the action is a freeform text field, that is defined by the developer. Consequently, it often is contaminated with unnecessary data, such as a user name, which should be filtered out.

The conceptual model of software operation data is depicted in Figure 2. It is arranged in three parts: the software artifact component, the architectural component, and the runtime component. The first three components cover static aspects of the software, whereas the latter, the runtime component, covers the dynamic aspect of software.

2.1.1 *Software Artifact Components*

A Function represents a small code unit that can be executed in source code, such as a method or a function. A Function is *defined* in some Container. Containers repre-

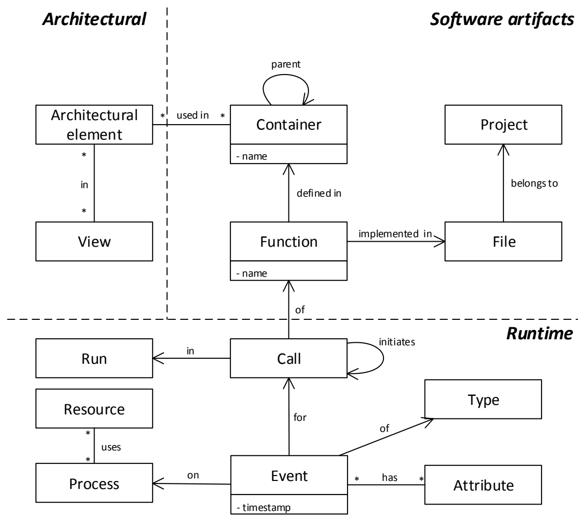


Figure 2: Conceptual model of software operation data for architecture mining

Table 1: Excerpt of an example run trace.

Datetime	Package	Process,	action
15-09-2016 03:30:02	[runner.core.start]	(7)	David Start
15-09-2016 04:36:05	[runner.core.run]	(7)	Run
15-09-2016 04:38:07	[runner.core.end]	(7)	David End
15-10-2016 03:36:15	[runner.core.start]	(29)	David Start
15-10-2016 03:42:05	[runner.core.run]	(29)	Run
15-10-2016 03:48:55	[runner.core.run]	(29)	Run
15-10-2016 05:54:45	[runner.core.end]	(29)	David End
15-10-2016 14:00:00	[runner.core.start]	(17)	Peter Start

sent larger units of source code, such as classes, packages, and namespaces. Typically, the names of the functions and all parenting containers together form the Fully Specified Name (FSN) of each function. Ideally, this FSN should be unique for each function within a software system.

Source code is divided into Files that are organized in Projects. Each Function is implemented in some File. In the conceptual model, we abstract from the folders that actually contain these files. This links the software operation to the actual development view of the software, allowing software developers to relate the operation to the projects in which the software is organized, allowing for software analytics [4].

2.1.2 *Architectural Component*

As described in the ISO standard 42010 [11], an architectural definition of a system consists of Architectural elements that are *contained in* views on the system. Within software architecture, many different viewpoints exist [14] that each focus on specific aspects of the software. Traceability [12] focuses on the mapping of logical Containers on Architectural Elements. Creating such a mapping is an important task of software architecture reconstruction [9], and essential for architecture conformance checking [6]. Maintaining such a mapping is a hard task in general. However, in order to close the gap from source code to architecture, it is essential. Many different techniques exist to maintain such a mapping, ranging from manually created documentation to annotation in the source code [15].

2.1.3 *Runtime Component*

The runtime component is the actual heart of the software operation data. It describes the actual software operation data. Within a Run of the system, be it the actual operation of the software system or test runs, such as regression or acceptance tests, Functions are being called and executed. Such a Call is *for* some Function *in* a Run. Different from approaches like MTF2 [1], we record the Events that occur for a Call. These events are *of* some Type, representing transitions of the life-cycle of a Call, such as the *Start* and the *End* event. Other events like *suspend*, *interrupt* or *resume* are possible as well, depending on the software system under investigation. An Event may *have* different Attributes, such as the parameters or objects that are passed. The execution of a Call is *performed on* one or more Processes that utilize some Resources. As a Call may be started on one process, get suspended and then resumed on some other process, Process is connected to the Event, rather than to the Call itself.

2.2 Framework

To support the architect in analyzing software operation data, we developed the Architectural Intelligence Mining Framework (AIM), which is an extendible framework comprising three types of components: parsers, analyzers, and visualizers. Its architecture is depicted in Figure 3. The framework is written in C# and builds upon the ASP.NET Core framework. It is backed by a Postgres and a Neo4j graph database.

2.2.1 *Parser Component*

The Parser component translates software operation data and populates the software operation data storage. As software systems log in their own format, each system requires a different parser. For every log entry, the parser creates an event. Traces are constructed based on some case id, such as a process id. Lastly, every trace is sorted chronologically by datetime to make sure that even when events are logged out of order, the traces reflect their true ordering. In some cases the process id is not available, so we group the instances on their thread id. This is not perfect, as multiple sequences running on the same system thread are seen as a single sequence. To mitigate this problem we further split the traces any time there is a preset time interval between two consecutive event instances on the same thread.

2.2.2 *Analyzer Component*

The output of the parser reflects the low-level events of the system, whereas architectural intelligence focuses on high-level models. To close this gap, the Analyzer components enhance the software operation data. Different Analyzers can be run consecutively on the software operation data to create better abstractions.

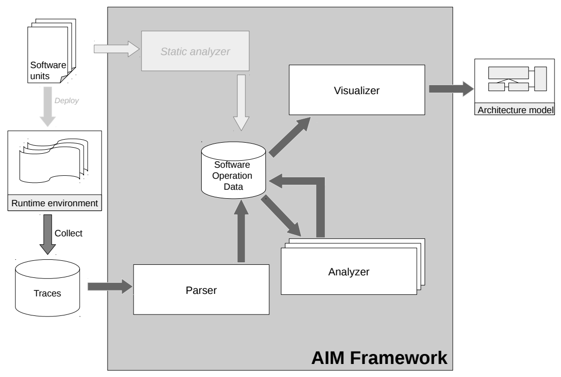


Figure 3: The AIM Framework Architecture

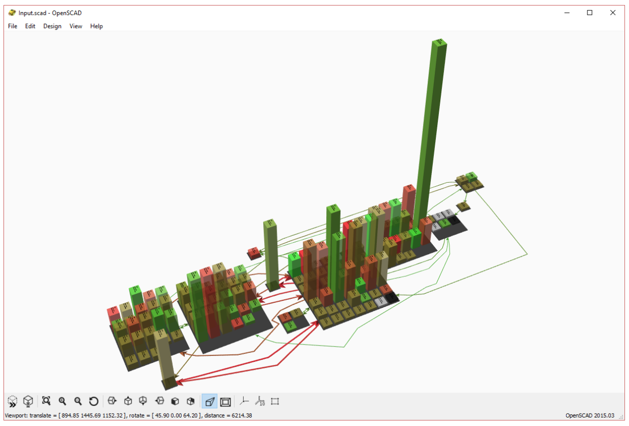
For example, whereas traces only reflect the partial order of events, abstraction techniques such as the communication behavioral profile [18] allow discovery of higher level relations between elements in the software operation data, such as causal relations between architectural elements. A second analyzer then clusters the architectural elements based on the discovered causal relations.

2.2.3 *Visualizer Component*

To visualize the results of the analyzers, different visualization techniques can be applied. In the next section, we will discuss our main visualizer: ArchitectureCity.

3. VISUALIZING RUNTIME AS A CITY

The most significant visualization component in our framework is ArchitectureCity. It combines dependency graphs, CodeCity, and the dynamic aspects of a running system captured in the software operation data. It is implemented as a visualizer within the AIM Framework and uses the output of clustering analyzers. ArchitectureCity takes the high-level intuitive overview and node information properties of CodeCity and combines it with the interactions between components that we know from a dependency graph. An example is shown in Figure 4.



Viewport: translate = [894.85 1445.69 1152.32], rotate = [45.90 0.00 64.20], distance = 6214.38

Figure 4: ArchitectureCity Visualization utilizing Package Clustering with package depth of 3

3.1 Visualizing as a City

As with CodeCity, buildings are pieces of functionality, but as we have no knowledge of the actual source code, we create a building for each unique log event. This visualization is coupled with Analyzers that implement clustering algorithms to provide city districts. A district is a cluster of buildings. Different clustering techniques can be applied, such as the hierarchy imposed by the FSNs related to the events, or modularity to maximize within-cluster interactions. The clustering algorithms allow for different clustering depths and, therefore, different levels of detail within ArchitectureCity. Each building belongs to a single district and each district contains one or more buildings.

The sizes and colour of buildings depend on various metrics of the events corresponding to the building. This allows us to display a range of metrics in a single view. As opposed to CodeCity, roads between districts are possible. These roads are directional and depict interactions between buildings from the connected districts. Relations are only represented between districts, not between buildings contained in the districts.

3.2 Dimensions

ArchitectureCity is able to visualize many dimensions of information at the same time through different visual aids. Both buildings and roads have a placement, colour, and size, depending on the corresponding event or interaction.

3.2.1 *Buildings*

Buildings are the most flexible, as they possess a placement, height, width, length, colour, and opacity. This would allow for six axis of information to be displayed at once. In its current state, ArchitectureCity buildings do not scale in width or length as this would require a new placement algorithm within districts and would probably decrease the clarity of the overall visualization. The height is based on the number of occurrences of the event within the log. As these values can range from one to tens of thousands, we chose the natural log of the number of occurrences as actual building height to reduce scaling issues. The colours can match the height of the buildings, or can show the fan in/out ratio. When displaying the latter, it uses a colour range from green to red corresponding to a high in/out ratio and a high out/in ratio respectively. Buildings with only outgoing roads are light-gray while buildings with only incoming roads are black. The opacity displays the absolute fan in/out value, with a more transparent building having a lower absolute fan in/out count.

3.2.2 *Roads*

Roads do not have as many dimensions but still feature placement, width, colour, and opacity. The placement of roads indicates which components are connected, and in what direction communication takes place. The colour and width depend on the strength of the interaction, in the current state it is based on the number of interactions. Alternatively, the roads can easily be modified to e.g. indicate the average time delay between actions or the volume of data passed between buildings.

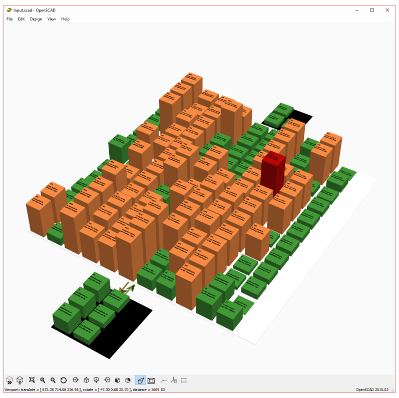
3.3 Realization

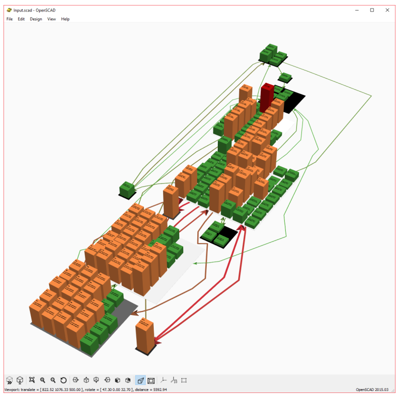
As a proof of concept, we implemented ArchitectureCity using the AIM Framework. For the visualization, we rely on both the Graphviz [1] package, as well as OpenSCAD [2]. The latter is a software package typically used for 3D-modelling and printing. The implementation first determines the districts based on a selected clustering algorithm. Next, it generates an output file with 3D information that can be parsed by OpenSCAD.

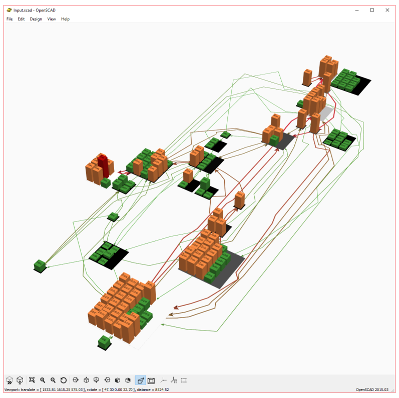
3.3.1 *Placement*

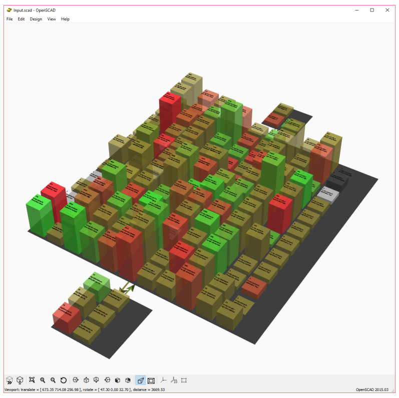
After completing the clustering, the placement of the districts and roads is computed by Graphviz DOT, which generates a .dot file containing all information to create a 2D placement. This forms the 2D base for ArchitectureCity, as it uses these exact coordinates. As Graphviz clustering is an experimental feature that is not fully implemented, we opted to create the clustering ourselves, letting Graphviz

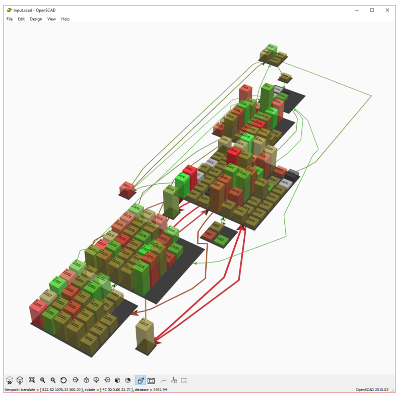
[1] www.graphviz.org/
[2] www.openscad.org/


(a) Depth 1


(b) Depth 3


(c) Depth 5


(d) Depth 1


(e) Depth 3

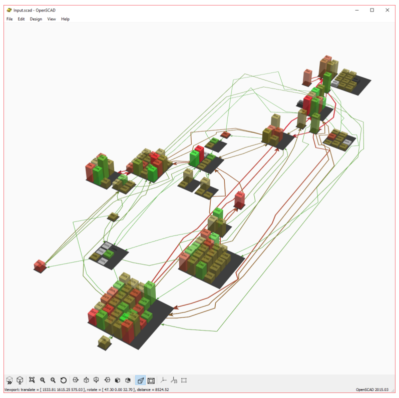

(f) Depth 5

Figure 5: ArchitectureCity for varying levels of package clustering, using call count-based colouring (top) and fan in/out colouring (bottom).

generate a placement for the districts and roads and fill in the buildings afterward. The size of the district corresponds to the number of buildings it contains to always have enough space to later place them. The district width and length are always equal to the square root of the number of buildings it contains; this does mean that it is possible to construct a district with an empty row. This occurs when the number of buildings within a single district is n where $n <= (\lceil\sqrt{n}\rceil - 1)\lceil\sqrt{n}\rceil$.

The output of the placement is a dot file containing all elements with their corresponding coordinates. This can be used to generate a 2D image of ArchitectureCity.

3.3.2 *Populating*

The Graphviz coordinates and sizes are parsed to the general building and road classes. This intermediate step was chosen to facilitate the support of multiple rendering programs with minimal effort. The building class only has information about the shapes, colours and, sizes of the 3D object, as well as possible text to be printed on it. The roads class contains the buildings it connects, the intermediate points between the buildings as computed by Graphviz, and the colour and width of the road. All colours and sizes of the buildings are computed in this step, as it is the last where we have access to all the event information. After this step, only the 3D object information is kept.

3.3.3 *Rendering*

After creating the buildings and roads, the visualizer parses the objects to the OpenSCAD format. Anti-Aliasing (AA) can be added by flagging OpenSCAD for GPU assisted Anti-aliasing. To make the images more visually appealing, AA was turned on in all images contained in this paper.

4. INITIAL EVALUATION

To demonstrate the capabilities of the AIM Framework, and ArchitectureCity in particular, we present a case study applied to an organization that handles delivery logistics in the Netherlands. The organization provided us with logs from six different systems, collected during three different days. We had no access to the source code of the system. The one-day intervals were chosen because the organization mostly operates within a weekly cycle with a one-day peak in processing. The logs are of the peak day when the system is most heavily utilized. The logs follow a similar structure as shown in Table 1. In total, the logs together contain 152.865 events. Unfortunately, there is no dedicated process identifier, the log only contains thread identifiers. We, therefore, chose the latter to relate actions within the log. Additionally, we implemented a time-based feature that splits the actions at a thread into traces.

In Figure 5, three visualizations of the obtained software operation data are shown. The visualizations use the package-clustering algorithm on different depths. A first observation is the clean, but also deep package structure. This suggests that the system is written in Java. Next, we see a taller building in the center package (Figure 5(c), coloured red), indicating that this feature is called very frequently. The fan in/out colouring additionally shows that this function has a higher fan/in than out, indicating that this is a central feature in the system.

Another observation is that Figure 5(c) has an isolated component in the upper-right corner: it has no arrow from or to it. This indicates that the logs actually comprises two different systems that have no interaction.

In the depth 1 visualization, depicted in Figure 5(a), we observe that most functionality is within the same top level-package and that there is little communication between top-level packages.

As a next step, we evaluated our findings with the architects of DL. They were immediately able to correctly label the tall building. Most components found by ArchitectureCity were correctly recognized. In one case there was a strange, above-average activity node that was inspected in more detail and the event information surprised the architects. This component was thought to be removed or deactivated but was still showing up in the logs as an active component. Insights such as these were perceived incredibly advantageous, and the model was called a "model of truths" rather than the intended models on which the application was based. All systems shown were correctly identified, and clusters and system flow were largely expected and true to specification. They were surprised by the amount of information their log contained and happily surprised by the visualizations we presented.

Clear benefits of the system were identified immediately. Some improvements were requested: a tool-tip, more information on the clusters and making buildings clickable to their source. Adding time-based features was also discussed. The generation of interval based slices of the log and visualizing each one yields a "living" 3D overview of real system usage. This was recognized as a fascinating feature that would only be possible through processing dynamic system information and, therefore, impossible to achieve using their current techniques. In conclusion, the visualizations shown exceeded the expectations of the architects, and many insights e.g. the thought to be removed component, sparked interesting discussions about the DL system. ArchitectureCity improved their understanding and insight into their own software with which they have had years of experience.

5. CONCLUSIONS AND FUTURE WORK

Software operation data captures the actual behavior of software. In this paper, we have presented an approach that takes this data as input for architecture mining. For this, we have developed the AIM Framework, and more specifically, ArchitectureCity, which uses the analogy of cities to visualize the runtime of software: buildings, representing individual architectural elements are grouped in districts based on different clustering techniques, and streets depict the traffic between the different districts. Additionally, the streets, buildings, and districts can be coloured according to different metrics.

Initial validation through a case study shows the potential of the proposed approach: we provided the architects with new insights on the operation of their software.

The approach still has many limitations. For example, we rely on the notion of a case, which is typically not clear in a software system. Currently, no static information is used. We envision that if the software operation data is enhanced with static information, visualization will be better guided and result in more accurate representations. Another limitation of the approach is the placement of buildings and districts. Currently, we rely on graph layout algorithms. Different strategies can be applied, such as density and modularity to gain better visualization results.

Currently, we do not use all dimensions. For example, the size of buildings can be used to depict fan in/out, and similarly, the size of streets can be used.

Software operation data includes time information. In the current visualizations, we abstract from time. Adding time as a fourth dimension, e.g. using a time window, allows for visualizing software use and its evolution over time.

Last but not least, more validation is required to evaluate the output of ArchitectureCity. Initial evaluation shows the potential of the approach, but more research is required.

6. REFERENCES

[1] L. Alawneh, A. Hamou-Lhadj, and J. Hassine. Towards a common metamodel for traces of high performance computing systems to enable software analysis tasks. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 111–120. IEEE Computer Society, 2015.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Series in Software Engineering. Addison Wesley, Reading, MA, USA, 2012.

[3] J. Buckley, N. Ali, M. English, J. Rosik, and S. Herold. Real-time reflexion modelling in architecture reconciliation. *Inf. Softw. Technol.*, 61(C):107–123, May 2015.

[4] R. P. Buse and T. Zimmermann. Analytics for software development. In *FSE/SDP Workshop on Future of Software Engineering Research*, pages 77–80. ACM, 2010.

[5] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[6] L. De Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[7] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. *Proceedings of the 23rd annual international conference on Design of communication documenting & designing for pervasive information - SIGDOC '05*, page 68, 2005.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[9] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, 2009.

[10] A. D. Eisenberg and K. De Volder. Dynamic feature

traces: Finding features in unfamiliar code. In *21th IEEE International Conference on Software Maintenance*, pages 337–346. IEEE, 2005.

[11] ISO. *IEC 42010-2011: Systems and software engineering – Architecture description*. Genève, Switzerland: International Organization for Standardization, 2011.

[12] P. Lago, H. Muccini, and H. van Vliet. A scoped approach to traceability management. *Journal of Systems and Software*, 82:168–182, 2009.

[13] L. J. Pruijt, C. Köppe, J. M. E. M. van der Werf, and S. Brinkkemper. Husacct: Architecture compliance checking with rich sets of module and rule types. In *ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 851–854, New York, NY, USA, 2014. ACM.

[14] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2011.

[15] F. Schmidt, S. G. MacDonell, and A. M. Connor. An automatic architecture reconstruction and refactoring framework. In *Software Engineering Research,Management and Applications 2011*, pages 95–111. Springer, Berlin, 2012.

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

[17] H. van der Schuur, S. Jansen, and S. Brinkkemper. Reducing maintenance effort through software operation knowledge: An eclectic empirical evaluation. In *15th European Conference on Software Maintenance and Reengineering*, pages 201–210. IEEE Computer Society, 2011.

[18] J. M. E. M. van der Werf and E. Kaats. Discovery of functional architectures from event logs. In *International Workshop on Petri Nets and Software Engineering (PNSE'15)*, volume 1372 of *CEUR Workshop Proceedings*, pages 227–243. CEUR-WS.org, 2015.

[19] J. M. E. M. van der Werf, R. S. Mans, and W. M. P. van der Aalst. Mining declarative models using time intervals. In *International Workshop on Modeling and Business Environments*, volume 989 of *CEUR Workshop Proceedings*, pages 227–243. CEUR-WS.org, 2013.

[20] J. M. E. M. van der Werf, C. v. Schuppen, S. Brinkkemper, S. Jansen, P. Boon, and G. van der Plas. Architectural intelligence: a framework and application to e-learning. In *EMMSAD 2017*, pages 95–102. CEUR-WS, 2017.

[21] J. M. E. M. van der Werf and H. M. W. Verbeek. Online compliance monitoring of service landscapes. In *Business Process Management Workshops, Revised Papers*, volume 202 of *LNBIP*, pages 89–95. Springer, Berlin, 2015.

[22] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[23] R. Wettel and M. Lanza. CodeCity: 3D Visualization of Large-Scale Software. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, page 921, New York, New York, USA, 2008. ACM Press.

[24] E. J. Weyuker. Evaluation techniques for improving the quality of very large software systems in a cost-effective way. *Journal of Systems and Software*, 47(2-3):97–103, 1999.

[25] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14. ACM, 2011.