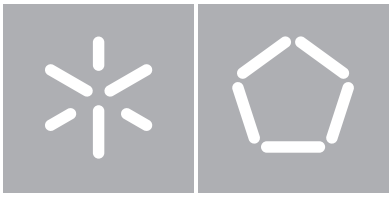
Universidade do Minho
Escola de Engenharia

Luís Diogo Monteiro Duarte Couto

Analysing call graphs for software architecture quality profiling

Junho de 2012



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Diogo Monteiro Duarte Couto

Analysing call graphs for software architecture quality profiling

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor José Nuno Oliveira

Junho de 2012

Acknowledgements

This dissertation could not have been written without the help of several fine people. In no particular order, I would like to thank:

- José Nuno Oliveira for supervising this project. His guidance and encouragement were invaluable.
- Manuel Alcino Cunha for first getting me interested in Formal Methods which eventually led to this dissertation.
- Joost Visser and SIG for the opportunity to work on this project, for assistance and suggestions on many issues and for supporting me financially on a conference trip.
- Eric Bouwers for graciously allowing the use of the IIOT model and assistance with the model analysis process.
- Miguel Ferreira for countless suggestions and constant assistance and advice on the project.
- Tiago Veloso for assistance with formatting and templates.
- A personal thanks to my family for their patience and support.

Abstract

Software has become ubiquitous in today's world. It is therefore obvious that software quality has become a chief concern of researchers and practitioners alike. Nowadays, Software Architecture is a core dimension of software quality. With the continuous increase in the size of software systems, its importance has grown further and further.

A good way to tackle quality from an architectural point of view is by focusing on complexity. Complexity is a primary reason for failure in software systems. We are particularly interested in a metric for complexity. One that is mathematically based and fundamentally sound. We present such a metric.

As software systems have grown in size and complexity, models have emerged as an excellent to way to analyse and reason about the architecture of a system. We present two architectural models: IIOT, which is focused on the organisation and structure of source code files and the connections between them; and SIP, which is focused on the higher level structure and functionalities of the software system.

This dissertation analyses and combines the two architectural models. We present a single unified model. This model has also been enriched with a metric for complexity. We also present a prototype tool that extracts architectural information from a software system's call graphs, builds the unified model for the system and calculates its complexity.

Furthermore, we perform an extensive analysis of the complexity metric by conducting tests on a large variety of software systems. We also explore the possibility of deriving functionality directly from a call graph via the application of clustering techniques.

Resumo

O software está em todo o lado nos dias de hoje. Por isso, é óbvio que a sua qualidade se tenha tornado numa grande preocupação quer para investigadores quer para praticantes. A arquitectura de software é hoje uma dimensão chave para a qualidade de software. Com o aumento constante do tamanho dos sistemas, a importância da sua arquitectura só tem aumentado.

Uma boa maneira de atacar a questão da qualidade dum ponto de vista arquitectural é concentrando-nos na complexidade. A complexidade é uma das principais causas de falhas em sistemas de software. Há pois todo o interesse em desenvolver métricas para complexidade bem fundamentadas na matemática. Iremos apresentar tal métrica.

À medida que as arquitecturas de software têm aumentado em tamanho e complexidade, os modelos emergiram como um bom modo de analisar e raciocinar sobre a arquitectura dum sistema. São apresentados dois modelos arquitecturais: IIOT que está focado na organização e estruturação dos ficheiros de código fonte e nas conexões entre eles; e SIP que está focado na estrutura alto-nível e ligações das funcionalidades do sistema.

Esta dissertação analisa e combina esses dois modelos arquitecturais num único modelo unificado que é enriquecido com uma métrica de complexidade arquitectural. Apresenta-se também uma ferramenta protótipo que extrai a informação arquitectural do grafo de invocação ("call graph") do sistema de software e que constrói o modelo unificado respectivo e calcula a complexidade do sistema de acordo com a métrica.

A métrica de complexidade arquitectural proposta é analisada extensivamente, através de testes e medições realizadas em vários sistemas de software. É também explorada a possibilidade de derivar as funcionalidades de um sistema directamente a partir do seu "call graph" com a aplicação de técnicas de classificação ("clustering").

Contents

List of Figures

List of Tables

Listings

Acronyms

AADL Architecture Analysis & Design Language.

ADL Architecture description language.

ATAM Architecture Tradeoff Analysis Method.

CB Component balance.

CSV Comma-separated values.

API Application programming interface.

IIOT Internal, Incoming, Outgoing and Throughput.

SIG Software Improvement Group.

LiSCIA Lightweight Sanity Check for Implemented Architectures.

LoC Lines of code.

OLAP Online analytical processing.

OO Object-oriented.

SA Software architecture.

SAAM Software Architecture Analysis Method.

SAR Software architecture reconstruction.

SIP Simple Iterative Partitions.

SOA Service-oriented architecture.

UML Unified Modeling Language.

Chapter 1

Introduction

1.1 Basic Motivation

Software has become such a crucial part of our lives that we depend on it in many ways (shopping, healthcare, banking, transportation, etc...). It therefore stands to reason that we are all interested that the software we use or develop is the best possible [20].

So we arrive at the first important notion in this dissertation, *software quality*, which has become a major concern in today's software community [38]. Quality means much more than simply writing good source code, it must be a part of the entire software process[33].

There are many facets and angles to software quality. These range from performance and "absence of bugs" to how easy it is to change the software (modifiability) or test it (testability). These different aspects of software quality are called *software quality attributes*.

There are, of course, many other aspects that are also relevant to the quality of a software system, including the quality of the source code itself. The software architecture (SA) of a system is a very strong factor for quality[6] because it *"allow[s] or preclude[s] nearly all of the systems quality attributes"* [19].

SA can be defined as *"the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."* [6].

It is also important to mention the measuring of software quality. As previously stated, software quality is reflected in a series of quality attributes. In order to measure the quality of a software system, we must establish the desired quality attributes and measure them. In order to measure a quality attribute we need metrics. The ones we use are usually called software quality metrics. It should be noted that not all quality attributes can be measured

directly. For example, the only way to directly evaluate the maintainability of a software system is to evaluate the time and resources spent by developers doing maintenance and not by measuring any metric on the system itself. However, it is possible to use metrics as predictors for certain given quality attributes. These metrics are referred to as metrics by proxy.

Our focus in on how to measure quality from an architectural complexity point of view. Therefore, we need a metric that relates to SA complexity.

1.2 Software Archicture Evaluation

As previously stated, we need a way to evaluate the architectural quality of a system. How good is the architecture? To what extend does (or will) it support the desired quality attributes? Clearly, we need metrics (and perhaps quality attributes) that are designed especially for SA.

Furthermore, SA is a fairly abstract concept and there are many models for representing and working with it. Each model tends to have its own focus or specialty. One of the most famous examples is Architecture Analysis & Design Language (AADL)[27] which focuses on embedded systems, software/hardware interactions and correctness.

This dissertation presents two architectural models: *Simple Iterative Partitions (SIP)* proposed by Roger Sessions [60] and *Internal, Incoming, Outgoing and Throughput (IIOT)* developed by Eric Bouwers et al. [13].

The SIP model puts it that complexity is an important part of architectural quality, though not the only one. SIP follows a principle that, everything else being equal, the less complex a system is, the better. SIP proposes both a model for representing the architecture as well as a metric (and respective formula) to measure the system's complexity. It is designed to work in the conceptual stages of software development. SIP typically evaluates an architecture in the design stages and its models use the system's functionalities as major elements. While this is a perfectly valid and useful approach, this dissertation aims to evaluate the architecture of implemented systems. This is where IIOT comes in.

The IIOT model focuses on implemented architectures taking a much more structural view of a system's SA. In fact, the IIOT model of a given software system can be extracted from its source code, which means we do not need to perform a functional analysis of a system. IIOT presents the architecture of a system resorting to a system's modules (generally, the various source code files) and components (sets of modules). IIOT defines a taxonomy

for classification of modules.

Both models have already been developed and validated. The main challenge is to combine them into a single, unified model. The idea is to harmonize both models and include them into a single process that, for a given software system, will allow one to reverse-engineer a model for the SA and then measure the quality of that architecture.

1.3 Architectural Information

SA is an abstract concept and therefore only implicitly present in a system. SA models allow us to visualize architecture although one needs a way to extract it. One must analyse systems and extract architectural information in order to build the models. This dissertation will focus on approaches that use static analysis. This is in line with the standard software evaluation procedure at Software Improvement Group (SIG). Many elements and data obtained from a software system could be used for static analysis. We will work exclusively with call graphs and extract architectural information from them. An advantage of using call graphs is that any system and language that features calls (as most do) can have a call graph. Another one is that call graphs contain everything needed to build SA models. This is a powerful abstraction of code that allows one to channel all resources and time into a single area, hopefully leading to more focused results.

Extraction, storage and processing of call graphs are all activities that must be kept in mind. We shall not be concerned with the extraction processes, as SIG has technology for doing so, kindly providing call graphs for experiments. Call graph information is fairly simple so its storage should not be a problem. Because of its simplicity, we look into a few concepts from data warehouses [17] and Online analytical processing (OLAP) [17] when considering storage. Processing the call graphs and using them to construct our models is obviously a very important part of this project.

1.4 Aims

This dissertation evolved from a number of research questions which are listed below, together with associated tasks. The prime objective was to unify two SA models (SIP and IIOT) based on a formal analysis.

Question 1: Can the SIP and IIOT models be unified?

Task 1.a: Formal analysis of terminology of the models.

Task 1.b: Modeling of a single, intermediate, representation.

The main hope behind the compatibility analysis of both models is to assess the utility of the complexity metric from SIP when used with the IIOT model. This is made implicit in another objective:

Question 1.1: Can we use SIP's metric with IIOT?

Task 1.1.a: Identify necessary elements for metric usage.

Task 1.1.b: Locate/add key elements for the metric to the IIOT model.

Task 1.1.c: Devise an algorithm for computing the metric with the IIOT model.

Before we can contribute anything to the field of SA, it is important that we gain a good understanding of the field itself and the research that has already been carried out. Towards that end, a survey of SA literature is carried out in order to answer the three following questions:

Question 2.a: What has been achieved thus far?

Question 2.b: What are the recent breakthroughs?

Question 2.c Which shortcomings remain?

The SIP model is built on a notion of business functions: independent pieces of functionality provided by a system. These are used in the complexity calculation for the system. In the SIP model, these business functions are constructed manually (for example, by analysing the design documents). On the other hand, the IIOT model is built automatically from the source code and so there is an interest in examining the relationship between source code and business functions, as shown by the following questions.

Question 3: Is it possible to derive business functions from static call graph analysis?

Question 3.a: Does it make sense to reduce the methods in a call graph to method groups?

Question 3.b: Is it within reach to obtain a "good" reduction?

Question 3.c: Can such analyses be performed in an automated manner on static gall graphs?

It was also decided to place great focus on the complexity metric by performing a detailed analysis of it. This is reflected in the following question and tasks. This was also used to power our case study and served as validation for the metric and related research done throughout the dissertation.

Question 4: What can we learn about the complexity metric when applied to IIOT instances?

Task 4.a: Perform a statistic analysis of the metric by itself.

Task 4.b: Analyse and compare the complexity metric with other software system metrics.

1.4.1 Clustering

The information presented in call graphs is ideally suited to constructing the IIOT model since both call graphs and IIOT essentially exist at the same abstraction level (implementation). However, the SIP model works with more abstract concepts such as functionalities (or functional groups). One of the ways in which we attempt to unify both models is by trying to extract SIP model elements from a system's call graph.

We are interested in ways to group the elements of a call graph (typically methods) into higher level elements (functional groups). We explore several approaches including graph theory and also graph clustering techniques.

Particularly interesting graph clustering techniques show up in the field of community detection [22]. We explore the area under the intuition that some of the work might be applicable in our project.

This is the more experimental aspect of our project with a great deal of trial and error and less expectations in terms of results.

1.4.2 Quality Metric

The main goal of this dissertation is to have a way to measure the complexity of a system from an architectural point of view. We extract the architectural information and use it with the IIOT portion of our work. The metric itself comes from the the SIP portion. Once we have combined both models, we are then able to apply the metric directly to a system.

To help with this we develop a prototype tool that implements the fundamental steps of the process: read a call graph, construct the model, compute the metric. With this tool we

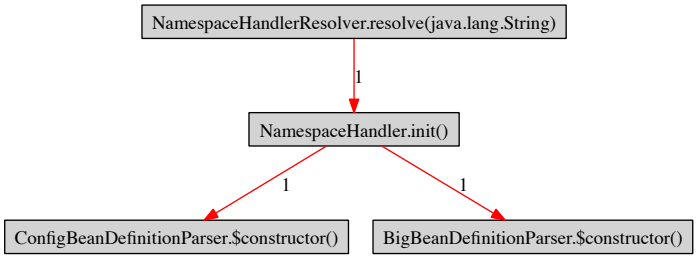


Figure 1.1: A (very) simple callgraph example.

can evaluate our work on several test systems. We are especially interested in seeing how the SIP complexity metric varies across different systems.

The main contribution of this dissertation is therefore a metric-based approach to measuring the complexity of a system's SA. We use a combination of design and post-implementation models and approaches, thus establishing a bridge among these. We also hope to, indirectly, contribute to the clustering of software call graphs.

1.5 Background Information

1.5.1 Introducing Call Graphs

A call graph is a directed graph that represents referential (calling) relationships among the elements of a software system. It is a (relatively) old and very well established concept [58]. In a call graph, the nodes represent an element of the system, traditionally a function or procedure. Each edge in the graph represents a call between a pair of elements. For example, edge (a,b) tells that element a calls element b.

There are two main types of call graph: dynamic and static. A dynamic call graph corresponds to a specific and particular execution of a program. Each dynamic call graph is constructed by tracking the execution of a given program. A static call graph, on the other hand, is independent of program execution. It contains all possible calls among elements. Static call graphs are usually extracted from the program source code.

A very simple example of a method-level call graph from a Java system can be seen in Figure 1.1 (in truth, only a subgraph of a much larger system).

In this dissertation, we work exclusively with static call graphs. There are two reasons

for this. First, we can extract static graphs directly from source code, without the need for program execution. This stays with the tradition in SIG's approach to software evaluation, which is based on source code analysis. Secondly, a static call graph has information about the complete system. A dynamic call graph only contains the units and calls of a particular execution and is therefore not adequate for the kind of architectural analysis we want to perform.

Call graphs can be used to represent information at many different abstraction levels of a system and with varying granularity. We can have the traditional function-level call graph although we can also have a graph for calls at class-level, file-level, package-level and so forth. From a practical standpoint, one typically extracts the lowest level graph (function, method, etc.) and then simply rolls up the nodes to reach the higher levels, as in data mining. Call graphs are generic (they simply have generic units and calls) though in this dissertation we work specifically with Java call graphs (with method calls and so forth).

1.5.2 Introducing Alloy

In this section we will briefly introduce Alloy, a modelling language used later on when we analyse IIOT and SIP. Alloy is a modeling language developed at MIT[37]. It is built upon first order logic and relational algebra (Alloy's lemma is *"everything is a relation"*). Its language is declarative and fairly simple.

Alloy has tool support in the form of the Alloy Analyzer which allows for execution and verification of the modules to search for counterexamples to assertions in the model and the consistency of the model itself (whether it can be instantiated or not). When counterexamples are found, the tool displays a diagram of the model for the user to inspect and explore.

In Alloy, relations are declared using the keyword `sig` (for signature):

Listing 1.1 A relation in declared in Alloy.

```
sig A {
  C : B
}
```

This declares a relation $C : A \rightarrow B$. Relations can be further defined with regards to multiplicity with keywords such as `lone` and `one`. Furthermore, a single sig may hold multiple relations:

Listing 1.2 Multiple relations declared in Alloy.

```
sig FileSystem {
  files: set File,
  parent: files -> lone files,
  name: files lone -> one Name
}
```

Relations can also be manipulated by several operators such as . (composition), & (intersection) and ~ (converse).

Finally, in order to express the various properties of a model, Alloy has predicates as shown in Listing 1.4.

Listing 1.3 Simple predicate in Alloy.

```
pred hasParent [f : File] {
  some f.parent
}
```

Predicates can then be used in assertions (`assert` command) to place constraints upon the model. These assertions can then be checked for a particular scope (number of instances of each relation in the model).

Listing 1.4 Checking an assertion in Alloy.

```
assert NoOrphans {
  all f : File | hasParent[f]
}

check NoOrphans for 3
```

1.6 Structure

The remainder of this dissertation is structured as follows:

- Chapter 2 gives an overview of the state of the art of SA research.

- Chapter 3 contains the analysis and unification of the two architectural models: SIP and IIOT.
- Chapter 4 discusses our experiences and attempts at applying clustering techniques to call graphs.
- Chapter 5 contains a presentation of the architectural complexity metric and a prototype tool used to help test our work.
- Chapter 6 presents the analysis or our work, by running the proposed metric on a series of test systems.
- Chapter 7 ends the dissertation with some thoughts about the final result and the process itself as well as some possibilities for future work.

Chapter 2

Software Architecture State of the Art Review

Software architecture (SA) has become a full-fledged discipline in recent years [61]. The field has come a long way from its earliest days when it was little more than a set of practices and methodologies for Object-Oriented programming. This chapter surveys research in the field of SA.

This chapter is structured as follows: *Section 2.1* gives a historical overview of SA and discusses major breakthroughs that have taken place in the field; *Section 2.2* reviews the state of the art in SA research and is the main part of the chapter; *Section 2.3* attempts to apply formal concept analysis to produce a structured view of the overall bulk of research in the field; *Section 2.4* discusses some of the present and future challenges for the field of SA; *Section 2.5* presents concluding remarks and discusses the outcome of this survey.

2.1 A brief history

SA traces its origins back the 80s [61]. Back then several people were already proposing the decomposition of a program in various modules and ways to look at those modules from a higher level of abstraction. At the same time, recurring software structures for particular domains began to appear (for example oscilloscopes [23]).

The term SA began to appear in the literature in the early 90s and became a research field of its own right. Two papers - [32, 56] - are generally credited with establishing the field and settling its name.

SA evolution cannot be dissociated from the success of the Object-Oriented (OO) paradigm.

OO programming puts a great focus on separation of concerns, encapsulation and structure. Soon people began talking about OO design and developing metrics for measuring the quality of an OO program [18]. These metrics were designed specifically to handle OO systems and measured unique OO aspects.

Soon enough, formalisms appeared for representing OO systems and concepts such as *Classes* and *Methods*. Several OO metrics were also formalized (such as cohesion [14] and coupling [15]) and the field began to move towards a focus on measuring a system's quality by analyzing its architectural aspects.

A major development was the advent of the Architecture Description Language (ADL). Traditionally, architects reasoned about a SA with simple "box and line" diagrams. ADLs go much further than that. They are actual languages supporting formalisms and are used to model and describe SA. ADL research grew tremendously and very soon several different ADLs came to exist, often competing for the same purpose [51].

With better formalisms in place, the evaluation, or analysis, of a SA became the main focus of research in this field. The main idea is to measure an architecture's ability to deliver on one (or more) specific quality attributes such as reliability, modifiability, etc. Several methods devoted to particular SA evaluations were developed by the community and surveyed in [5, 25].

Lack of tool support, a problem in the earlier days [39] has begun to subside in more recent times. Quite a few evaluation tools exist today, see for example [3].

Most evaluation methods originally focused on the designed architecture since it exists earlier in the development process and the cost of changing should in principle be lower. However, more work has recently been done on implemented architecture evaluation as part of an overall trend of software quality analysis [12].

A particularly elegant concept that eventually took shape was that of a *view*. A SA can be very complex and affect multiple domains, for example, hardware, source code, protocols and networking. Architects would often try to cram everything into a single, overly complex diagram. The notion of views argues that a single SA can be viewed and analyzed from multiple separate but interconnected and consistent viewpoints. Each can be represented by its own diagram. Particularly influential was the "4+1" view model [42] which proposes four viewpoints on a system: *logical*, *process*, *physical* and *development* plus a fifth.

Interestingly, the "+1" view, scenarios, became widely adopted in several evaluation methods leading to so-called scenario-based evaluation [5]. This also ties into the popular concept of an use case, as a scenario is an instance of a use case [42].

Clearly, SA has come a long way and is now a proper research field, although still fairly young and with much untapped potential.

2.2 Fields of software architecture research

We now discuss more in depth about some of the current branches of SA research. There are, of course, many different areas of research. We discuss some of the more important ones.

2.2.1 Software Architecture Evaluation

Evaluation is one of the most important areas in the field of SA research. Evaluation, or analysis, of a SA aims to assess its quality or inherent risks [5, 25].

Quality is generally expressed through several quality attributes (such as modifiability, flexibility, maintainability, etc.). Evaluation typically tries to ensure that the required quality attributes are met by the architecture while also identifying potential risks.

Much work has been done on the development of specific methods for evaluating a SA [5, 25]. Most methods advocate that analysis be performed in the earlier stages of a software project. The rationale is that costs of correcting problems are smaller in earlier stages.

As such, these analysis methods can be said to target a designed architecture. They try to predict the quality or risk of the final system through analysis of the SA. Obviously, the degree of precision in these methods can vary.

On the other hand, methods that analyse an implemented architecture also exist. These methods target an architecture post-implementation and most often from the point of view of the source code. These methods have an advantage in the sense that tools can be developed to automate some of their activities (e.g. static code analysis). The main disadvantage is, obviously, that changing something is much more costly at this stage and any problems uncovered will typically be more expensive to correct than if they had been uncovered earlier on.

Nowadays most software systems have a fairly long lifespan and continue to change and evolve through their lifetime. As a system continues to evolve, it can develop several architectural problems such as *drift* [56, p. 43] (the system's implementation becomes less and less adherent to the designed architecture); *erosion* [56, p. 43] (the implementation specifically violates the architecture); or *mismatch* [29, 30] (various components of the system do

not properly fit together). These problems can also tie into the concept of *technical debt* [16] - situations where long-term source code quality is sacrificed to achieve a short-term gain in another area. Technical debt and architectural issues quite often go hand in hand.

Methods that focus on implemented architectures can be employed to detect architectural problems and ensure that they are corrected as the system evolves.

To better contribute to the discussion of SA evaluation methods we begin by presenting two important methods in more detail: the first evaluation method to be developed and one of the most popularly used today.

Software Architecture Analysis Method (SAAM) [40]

SAAM was the earliest method developed for SA analysis. SAAM's main goal is to identify the risk associated with a particular SA. In terms of quality attributes, it measures *modifiability*. SAAM uses as input requirements documents and an architectural description. This description relies on three views of the SA: functionality, structure and allocation of the former to the latter. SAAM's primary method of evaluation is based on scenarios that introduce change (hence modifiability). SAAM has since been adapted and modified to evaluate a SA on other criteria such as *reuse* [46, 52] and *complexity* [44]. Interestingly, SAAM for evolution and reuse [46] is an example of a method that targets an implemented architecture.

Architecture Tradeoff Analysis Method (ATAM) [41]

ATAM was developed under the idea that the quality attributes of a software system interact with each other. For example, performance and modifiability affect each other. As such, a method that analyzes an attribute in the void might not, in fact, give a proper assessment of the system. ATAM can analyze a SA in relation to multiple attributes and identify trade-off points among them. ATAM relies on the "4+1" model to describe architectures. It uses multiple evaluation techniques such as scenarios and also questions and measures. ATAM is regarded as a well validated and comprehensive method [5].

Several other methods exist such as Architecture-Level Modifiability Analysis (ALMA) [9], Performance Assessment of Software Architectures (PASA) [65], Software Architecture Reliability Analysis Approach (SARAH) [63], Scenario-based Software Architecture Reengineering [8] and Simple Iterative Partitions (SIP) [59]. All these methods vary in goals and applicability. Many of them have reached solid levels of maturity [5]. However, according to a recent survey [4] adoption of these methods is still very low at the industry level, with some exceptions. SIP, for example is used by a consulting firm.

Reference [36] surveys methods for reliability and availability prediction at the SA level, two crucial quality attributes in today's software systems. They find that no single method is sufficient for predicting reliability and availability. Among the main shortcomings found are lack of tool support and poor validation of the methods.

One final method worth mentioning is Lightweight Sanity Check for Implemented Architectures (LiSCIA) [12]. One of LiSCIA's strengths is ease of use as it is designed to be usable out-of-the-box. LiSCIA targets implemented architectures and is aimed at preventing architectural erosion.

Additional research into utilizing SA to support quality attributes has also been carried out. For exemple, in [7] the authors investigate how SA can affect usability. They argue that traditional methods for achieving high usability focus almost exclusively on the UI. However, they argue that many usability aspects (for example, the undo functionality) require support from the architecture. They present several usability scenarios that require architectural support and also an architectural pattern that satisfies the scenario. In the end, they conclude that a strong link exists between SA and usability.

Research that directly targets quality attributes is also underway, for instance, work on the relationship between service-oriented-architecture (SOA) and different quality attributes, in particular on how SOA affects each of these attributes [54].

Another interesting area of research related to SA evaluation is software architecture reconstruction (SAR) [26]. These techniques attempt to rebuild a system's architecture in the context of maintenance tasks. Since the architecture is not explicitly present in the system and evolves over time, it is important to rebuild it in order to analyze it.

2.2.2 Architecture Description Languages

Almost any activity related to SA some way to define and work with the architecture. Originally, this was done in an *ad hoc* manner with "box and line" diagrams. These days, ADLs are commonly used in many software projects. They are particularly useful in the design phase. We shall begin by presenting two radically different ADLs.

AADL [27]

Originally known as Avionics Architecture Description Language, it was created for modeling software systems in the aviation field. It has an obvious focus on embedded systems though it can be utilized in other contexts. AADL allows one to model both the software and hardware portions of a system. An AADL model can be used as de-

sign reference, to support architecture evaluation and even allows for code generation. AADL is a formal language with a precise semantics. It is typically used for developing critical systems.

Unified Modeling Language (UML) [28]

UML has emerged as the *de facto* standard for SA modeling and has wide industrial adoption, in spite of being far from universally acclaimed [61]. Its strengths are excellent tool support, ease of usage (at least compared to something like AADL) and a fair amount of flexibility. Its weaknesses are a direct consequence of its strengths. UML is not a formal language and its diagrams are always liable to subjective interpretation. This can complicate matters when one of its purposes is precisely to communicate information. In fact, the different ways in which UML can be used is a source of problems [43, 53].

It can be argued that UML and AADL are at opposite ends of the ADL spectrum. One is strictly formal and requires a significant commitment from practitioners in order to be successfully used in a project. The other is strict at best although it can be utilised in a project with great ease (and clearly, using some ADL support when dealing with the architecture is better than using none). Most ADLs tend to lie somewhere along this spectrum, though the ones surveyed tend to lean more on the AADL side.

Many ADLs also exist with a specific focus on some particular area, as is the case of Wright [2] which focuses on interaction and connection between components and has connectors as first class citizens; Rapide [45] which focuses on concurrency, synchronization, events and uses partially ordered event sets as an underlying formalism; or Darwin [48] which focuses on distributed systems.

Several other ADLs exist [51], with varying purposes, tool support and so forth.

One final ADL worthy of note is Acme [31]. Acme is a generic ADL which is frequently used to act as an intermediate format between other ADLs.

2.3 Formal Concept Analysis

In this section we apply formal concept analysis to SA scientific literature. The idea is to get a more structured overview of the research and maybe draw some interesting conclusions. Also, this can be thought of as an experiment into using concept analysis in the context of

literature organization. This approach has been expanded and led to a paper [21] presented at a conference.

To begin with, a very simple explanation of concept analysis will be given (better introductions to the topic exist elsewhere [66, 57]). Formal concept analysis is a way of automatically deriving an ontology from a set of objects and their properties (attributes). Generally speaking, from a matrix of objects and their attributes, one derives relations between sets of objects and the sets of their shared attributes. There is a one-to-one relation between each set of objects and each set of shared attributes. The pairs in this relation are called *concepts*. There are three types of concept: object and attribute; object only; attribute only.

A set of concepts obeys the mathematical definition of a lattice and can thus be called the *concept lattice*. This becomes particularly useful in terms of visualising information. An

Object	**Attribute**
Book1	Free
Book1	Hard Copy
Book2	Digital Version
Book2	Hard Copy
Book4	Digital Version
Book5	Available from Library
Book5	Free
Book5	Hard Copy
Book5	Digital Version
Book6	Free
Book6	Hard Copy

Table 2.1: Example of object and attribute matrix.

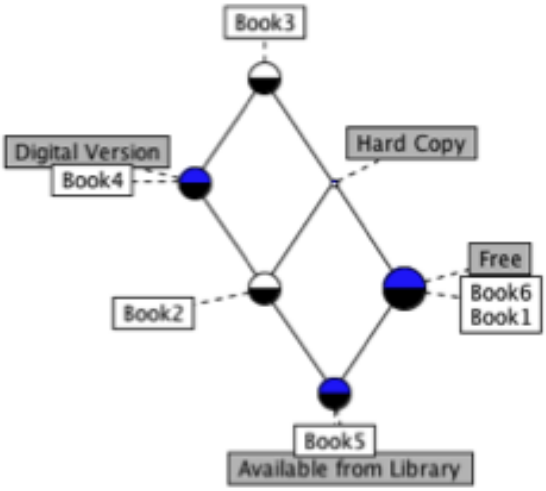


Figure 2.1: Example concept lattice

To support our work, we use a tool called Con Exp [1]. Cone Exp allows the user to supply the matrix and it will automatically compute the lattice. The lattice can then be visually explored and resized or reorganised as necessary. The lattice in Figure 2.1 was drawn with Con Exp. A quick explanation of the various visual elements follows:

- Grey labels represent attributes
- White labels represent objects
- White and black circles represent concepts inhabited by objects
- Blue and black circles represent concepts inhabited by objects and attributes
- Smaller white circles represent concepts that are empty or inhabited only by attributes

In order to perform concept analysis on the surveyed literature, each paper was classified with attributes from a predefined set. Each attribute in the set identifies a particular aspect of SA research. Developing a good set of attributes is essential. The attribute set for this purpose is as follows.

ADL Paper directly mentions an ADL (through definition or analysis).

Design Paper approaches the notion of design either from an OO or SA standpoint.

[1] http://conexp.sourceforge.net/

Designed Architecture (DA) Paper deals with SA from the design standpoint.

Evaluation Paper refers to evaluation of an SA generally with regards to quality.

Formalism Paper utilizes a formalism/formal method as the basis for some of its work.

Implemented Architecture (IA) Paper deals with SA inferred from source code.

Metrics Paper refers to metrics or measurements, generally with respect to SA and its quality.

Quality Paper refers to SA quality, generally in the form of quality attributes.

Reuse Paper refers the reuse of software components and its impact on SA.

Scenarios Paper refers to the usage of scenarios, generally as part of SA evaluation.

Tool Paper mentions a tool used to support/apply the research that was performed.

Views Paper mentions the notion of views as a way to look at an SA.

Figures 2.2 and Figure 2.3 present two versions of the concept lattice of surveyed papers and selected attributes. Figure 2.2 shows the lattice with citation keys as object labels whileFigure 2.3 shows the lattice with object counts.

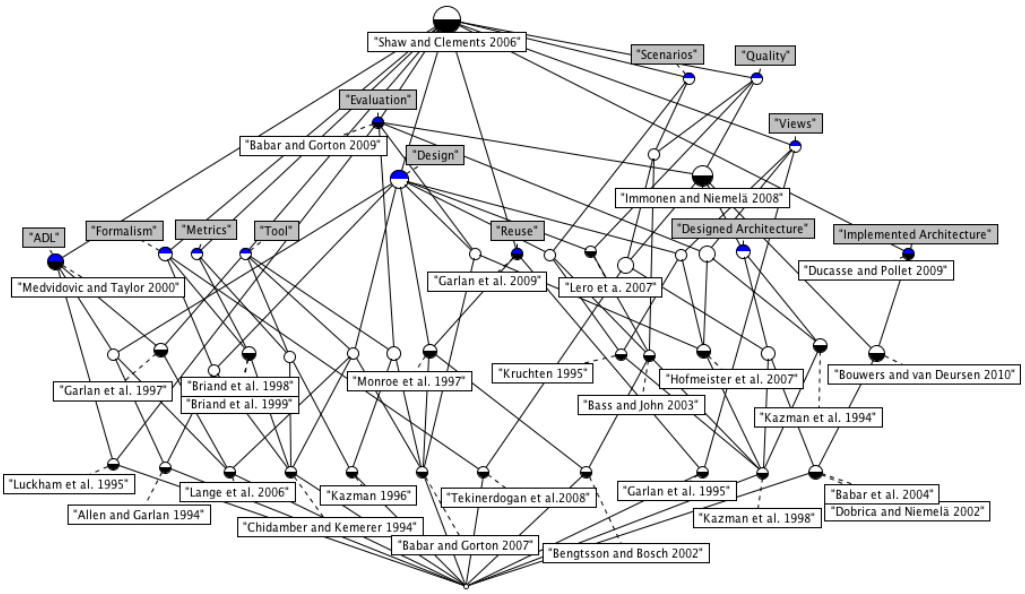


Figure 2.2: Concept lattice with citation keys and attributes

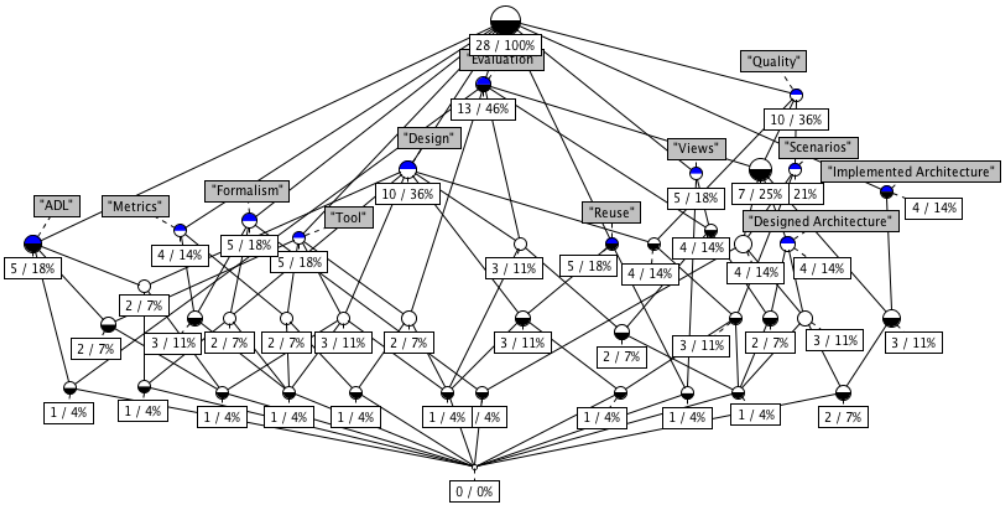


Figure 2.3: Concept lattice with extent object counts

The resulting lattice is visually complex and it is not easy to glean information from it. However, tool support allowed for easy exploration of the lattice.

Next we list some of the most relevant findings. We begin by examining how the attributes distribute themselves across the lattice. Later we will add objects to our considerations.

- There is no attribute in the topmost concept (an empty set of attributes). This means that no one attribute is shared by all papers.
- All attributes except one (*DA*) are independent from the others. The concepts they appear in descend directly from the root concept. This indicates that each attribute is its own “independent” field of study.
- *DA* descends from *Evaluation* and *Quality*. This indicates that some research on designed SAs is focused on evaluating the designs with regards to quality. This situation can be further explained by the existence of the *Design* attribute which does not descend from *Evaluation* and *Quality* and is used to classify research on the topic of actually designing architectures (versus evaluating designs). Interestingly, both *Design* and *Designed Architecture* intersect before reaching the bottom concept which means there is some research combining both the design and evaluation of SA. There

is one paper in this situation: “Kazman et all. 1998” [41]. This is obviously an evaluation method with a focus on quality attributes. But also, as the paper itself says: “The ATAM is a spiral model of *design*”. So it indeed covers both design and evaluation.

Nothing of interest can be further deduced from merely looking at attributes. We now begin contemplating objects as well.

- There is one object inhabiting the topmost concept. This is a paper that relates to no attributes. This may suggest that the attribute set is not comprehensive. The paper in question is “Shaw and Clements 2006” [61]. This paper gives a historical overview of SA and could almost as easily have been classified with every single attribute. It is therefore not a bad reflexion on the attribute set. However, a case may be made for removing the paper from the object set.

- Only four concepts have both an attribute and an own object. This indicates that much of the surveyed research cuts across multiple areas. The papers in question are: “Medvidovic and Taylor 2000” [51] (*ADL*); “Garlan et al. 2009” [30] (*Reuse*), “Babar and Gorton 2009” [4] (*Evaluation*) and ”Ducasse and Pollet 2009” [26]. Three of the papers are surveys on an area ([51] surveys ADLs, [4] surveys the state of practice in SA evaluation and [26] surveys SA reconstruction). Reference [30] is a very small article revisiting a previous paper. Once again a case may be made for removing it from the object set.

- Almost all objects inhabit their own concept. This indicates that few papers completely overlap. The ones that do overlap are: “Briand et al. 1999” [15] and“Briand et al. 1998” [14] (*Formalism* and *Metrics*) which makes perfect sense since both papers are very similar in everything except for the metric they treat (coupling and cohesion respectively); and “Babar et al. 2994” [5] and “Dobrica and Niemelä 2002" [25] (*Evaluation*, *Quality*, *Scenarios*, *DA* and *IA*) which makes sense since both are surveys of the exact same thing (SA evaluation methods).

- The paper with the most attributes (6) is “Kazman et al. 1998” [41] (paper that introduces ATAM). Most objects have 3 or less attributes which might indicate that the paper has too many attributes and might be worth re-examining. A review of the paper shows that it indeed covers all 6 areas. This is because ATAM is both used in terms of design and evaluation of an architecture which means it touches on most attributes of both areas. The paper is therefore properly classified.

- Two papers (“Babar et al. 2994” [5] and “Dobrica and Niemelä 2002" [25]) have 5 attributes. As mentioned above, both are surveys on SA evaluation methods. But their high attribute count justifies re-examination. In both cases, the attributes are *DA*, *IA*, *Evaluation*, *Quality* and *Scenarios*. All of them are related in some way to SA evaluation methods. And since that is the exact topic surveyed in those papers, their attributes make perfect sense.

- *Quality* and *Evaluation* are the attributes that have more objects in their extents (the set of objects that can be reached by descending paths from the node). This reinforces the idea that most research done in the area of SA aims at achieving software quality, often through evaluation. Indeed, *Evaluation* is the attribute present in the most objects, showing its size and importance in the field.

- Of the extent of *Evaluation* (13 objects), only 5 are in the extents of *IA* and *DA*. This suggests that either evaluation can be a field of research by itself (not targeted at designed or implemented architectures) or that some *Evaluation* papers need to be reviewed and their attributes rechecked. Upon re-examination of the said papers, we find that several of them indeed talk about SA evaluation from other viewpoints, which makes perfect sense. If all *Evaluation* papers also related to *DA* or *IA*, than a case could have been made for removing *Evaluation* from the attribute set.

2.4 Challenges

In this section, having reviewed the literature, we discuss a few challenges that remain in the field that we find particularly interesting.

The single greatest challenge is finding ways to increase productivity in most SA-related activities. In today’s software development world, productivity is chiefly important and notions such as agile methods are increasingly popular. SA activities such as evaluation or ADL supported design can be quite time-consuming and grind projects down. This makes them undesirable from an agile viewpoint.

On the other hand, the focus of agile methods on delivering functionality, sometimes to the exclusion of other important aspects, can lead to higher levels of technical debt and other architectural problems.

The challenge is to find ways to accelerate SA activities and find ways to incorporate them in modern development methods. There is particularly good research potential in this

area and plenty of work is already being done [1].

Tool support is already solid although it needs to improve even further. UML's success cannot be dissociated from its tool support. In order to continue the dissemination of SA practices developed in the academic field across the industry, there is greater need for tool support. More than developing academic tools, work must be done on proving that the development and support of comercial SA tools can be a viable business venture.

Another interesting area is quality. As systems continue to grow in lifespan, SA analysis becomes crucial to prevent erosion. In this regard, evaluation of implemented architectures is a rich avenue of research. Particularly, if combined with the aforementioned tool support.

Finally, the notion of a view. There are currently too many ways to visualize a SA. Ways to harmonize multiple views and models would be useful. This is obviously a big part of the theme of this dissertation.

2.5 Summary

We have surveyed the literature on SA and organized most research and major developments chronologically. We have also identified the main fields of SA research: evaluation and design (through ADLs). These can be considered the answer to our first question (What has been achieved thus far?)

Recent breakthroughs (our second question) include work on quality attributes for large-scale applications (such as reliability and availability) and attempts to unify SA and agile methods.

Major limitations (the third and final question) include a lack of first class tool support, a less than ideal standard ADL and remaining incompatibility with current developmental practices such as agile methods. Work is being done on tackling all these problems and more.

Chapter 3

Two Architectural Metamodels

In this chapter we analyze our two architectural models (IIOT and SIP) by providing abstract (meta)models of them using Alloy[1] for abstraction and analysis. The goal is to gain increased knowledge and familiarity with both models, check their mutual compatibility and define how to bridge them together.

3.1 IIOT Model

The IIOT model was developed internally at SIG [11]. Its main purpose is to model a SA from a viewpoint that is fairly close to the source code and that reflects the interactions between the various parts of an application.

3.1.1 Model Description

IIOT models a SA in terms of *components* and *modules*. Components can be seen as the "main (functional or technical) parts" of an application. Components are made up of modules (e.g. source code files). Each module can only belong to one component.

Modules expose their functionality through an application programming interface (API) that can be accessed from the module itself or from other modules. It is by analysing calls to the API of a module that we classify the modules in a component. Modules are classified as one of the following.

Internal Modules: are not called by and do not call modules from other components.

[1] The complete source code for all the Alloy models in this chaper can be found in Appendix A

Incoming Modules: are called by modules from other components but do not call modules of other components.

Outgoing Modules: call modules from other components but are not called by modules of other components.

Throughput Modules: call and are called by modules from other components.

The module classification is combined with their lines of code (LoC) volume in order to partition the component according to the distribution of code by each type of module.

3.1.2 Alloy Analysis

Listing 3.1 Alloy model for IIOT.

```
abstract sig Module {
  volume : one LoCVolume,
  mod_comp: one Component,
  calls: set Module
}

sig IntMod, IncMod, OutMod, TputMod extends Module {}
sig Component {}
sig LoCVolume {}
sig IIOTModel {
  modules: Module
}
```

This is a fairly simple model. We have two fundamental elements: `components` and `modules`. A `component` consists of several `modules` and these two entities are tied together by the `mod_comp` relation. It is worth noting that this relation connects each `module` to one and only one`component` and its converse connects a `component` to a set of `modules`. As for the `modules` themselves, we used a simple inheritance mechanism to define the various types of modules that IIOT contemplates. There is also a `calls` relation that, obviously, defines which `modules` call which.

The construction of this model was straightforward. It was simply a matter of defining all the necessary sigs which came directly from the model definition. Later it was necessary to develop a series of predicates to express the various rules which were once again extracted from the model documentation.

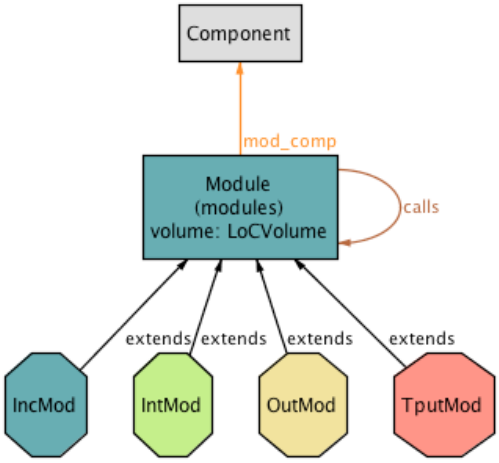


Figure 3.1: Alloy metamodel for the IIOT model.

Interestingly, when defining the call rules for each of the module types, it was necessary to explicitly state not only the calls that a module must have but also the calls it cannot have. In the end, we defined 4 base predicates that define each of the 4 call types. They are shown in Listing 3.2.

Predicates `NoCallsOutside` and `NotCalledOutside` define the types of call that a module cannot have, either outbound or inbound. In the case of `NoCallsOutside` this is done by forcing all the called modules (`mod.calls`) to be a part of the same component. These are given by the `fellows` function: `(mod.mod_comp)` gives the module's component while composing that with `mod_comp` gives the modules of the component. As for `NotCalledOutside` it uses the same rationale although applied inversely, ie, the modules that call the module in question belong to the same component.

The other two predicates (`CalledOutside` and `CallsOutside`) use a similar logic only forcing this time the called/calling modules to *not* be in the same component.

All these predicates are generic and can be applied to any `module`, though they only check a single module. Additional predicates that enforce them across the entire model are needed. They are shown in Listing 3.3.

3.1.3 Discussion

To begin our discussion of IIOT we present a list of related terminology.

Component: the highest level partition of a software system in the IIOT model. Components

Listing 3.2 General call rules for the IIOT Alloy model

```
pred NoCallsOutside[mod: Module]{
  mod.calls in fellows[mod]
}

pred NotCalledOutside[mod: Module]{
  calls.mod in fellows[mod]
}

pred CallsOutside[mod: Module] {
  some mod.calls
  mod.calls not in fellows[mod]
}

pred CalledOutside[mod: Module] {
  some calls.mod
  calls.mod not in fellows[mod]
}

fun fellows[mod : Module] : set Module{
  mod_comp.(mod.mod_comp)
}
```

are made up of modules.

Module: the lowest level partition of a software system in the IIOT model. A module typically represents a source code file.

Module call: a relation between two modules, extracted from a software system's call graph.

Outbound call: the module call relation from the point of view of the calling module. Modules "make" outbound calls.

Inbound call: the converse of the outbound call relation. Modules "receive" inbound calls.

Internal module: a module that only calls or is called by modules from the same component.

Incoming module: a module where at least some inbound calls relate with modules from other components and whose outbound calls only relate with modules from the same component.

Listing 3.3 Module type call rules for the IIOT Alloy model

```
pred Internal_Calls[iiot: IIOTModel]{
  all intM : IntMod | NoCallsOutside[intM] and NotCalledOutside[intM]
}

pred Incoming_Calls[iiot: IIOTModel]{
  all  incM : IncMod | NoCallsOutside[incM] and CalledOutside[incM]
}

pred Outgoing_Calls[iiot: IIOTModel]{
  all outM : OutMod | NotCalledOutside[outM] and CallsOutside[outM]
}

pred Throughput_Calls[iiot: IIOTModel]{
  all tputM : TputMod | CalledOutside[tputM] and CallsOutside[tputM]
}

pred All_Preds[iiot: IIOTModel]{
  Lone_Component[iiot]
  Internal_Calls[iiot]
  Incoming_Calls[iiot]
  Outgoing_Calls[iiot]
  Throughput_Calls[iiot]
}
```

Outgoing module: a module whose inbound calls only relate with modules from the same component and where at least some outbound calls relate with modules from other components.

Throughput module: a module where at least some inbound and some outbound calls relate with modules from other components.

Orphan module: a module that belongs to no component.

Inspecting several instantiations of the model unveils some interesting aspects which are worth discussing.

For instance, when classifying each of the module types there was a need to define both the types of calls a module can and cannot make.

There are several instances of modules calling themselves. While there can be some debate on the need to model these kinds of situations, it seems perfectly acceptable for them to exist.

Modules with no calls (inbound or outbound) are forcibly of type Internal. This is directly related to the definitions for the other types which all require the presence of calls. Only internal modules can have no calls and still respect their definition.

Another interesting point is that of *orphan* modules, which are modules that belong to no component. IIOT does not permit the existence of orphan modules. Though the model initially allowed for them, it was a simple matter to create an additional predicate to remove them.

In the end, it can safely be said that IIOT is a well built model. It is simple to understand (particularly since it exists at an abstraction level that is so close to the source code) and has no apparent contradictions. This understanding of the model seems more than sufficient to build a tool that incorporates it.

3.2 SIP Model

The SIP model was developed by Roger Sessions. It is used as part of a consulting business and its main purpose is to help reduce architectural complexity [60]

3.2.1 Model Description

SIP models the architecture of a software system from a functionality point of view. SIP is as much a process and a methodology as it is an architectural model. However, our purposes are focused strictly on the model. Additionally, though SIP is meant to model more than the SA (all the way up to the whole enterprise architecture) we simply use it for SA modeling. Because of this, we focus on a special version of SIP, which specializes in SOA. This version of SIP can easily be used to model a system's SA.

The main elements of the SIP model we used are:

Component The topmost partition of a system. Systems are broken down into several (top-level) components.

Functional group Represents a piece of functionality of the system (e.g. Make Payment or Login).

As mentioned above, the primary purpose of SIP is to reduce SA complexity. A system is decomposed and organized into components. Relations in the form of dependencies exist between components. These dependencies manifest themselves when a functional group from one component depends on a functional group from another component. In this case, the first component depends on (or is related to) the second. We generally call these relations connections.

After the system has been modelled as a series of functional groups, the process of reducing it by partitioning can begin. Partitioning typically involves breaking a functional group into lower level ones. These functional groups are then grouped according to synergy (in other words, the "most similar" functional groups are placed together). Afterwards, these groups of functional groups are packed into the various components of the system. Finally, we establish the necessary connections between the various functional groups and, by extension, components.

The proposed advantage of using SIP to assist in designing a system is that it leads to a system with the smallest amount of functional groups, components and connections. This in turn leads to lower complexity. This is because SIP argues that complexity is a function of the amount of functional groups and connections in the system.

SIP also includes a metric for measuring architectural complexity. This metric measures complexity as a function of the number of functional groups in each component and the number of connections between components. The metric is computed on a per-component

basis and then summed up to obtain the overall complexity of the system. This metric has an underlying mathematical basis and will be presented in chapter 5.

3.2.2 Alloy Analysis

It should be clear by now that SIP has a much greater focus on process than on the actual architectural model. In order to deepen our understanding of how an SA can be modelled in SIP we once again perform an analysis with Alloy. This gives us a better idea on how a SA looks like in SIP.

Listing 3.4 and Figure 3.2 represent a preliminary version of our Alloy model for SIP:

Listing 3.4 Alloy model for SIP.

```
// -- Sigs

abstract sig SIPSystem {
  fgs: set FunctionalGroup,
  components: set Component
}

sig FunctionalGroup {
  depends: set FunctionalGroup
}

sig Component{
  cnx: set Component,
  implements : set FunctionalGroup
}
```

As can be seen from the metamodel of Figure 3.2, the SIP model is a very simple one where a system is composed of components which are connected to each other via the `cnx` relation. In addition, there are various functional groups (modelled by `FunctionalGroup`) that are packed into components. This is modelled with the `implements` relation. `FunctionalGroups` are also connected to each other, as modeled with the `depends` relation.

There is little in terms of predicates worth discussing in this model. This is because the model has no taxonomy or classification and therefore, has very few rules to follow. In fact, all the predicates written for this model were mostly related to ensuring the various relations behaved correctly (for example, there is a predicate to ensure that a component is not connected to itself).

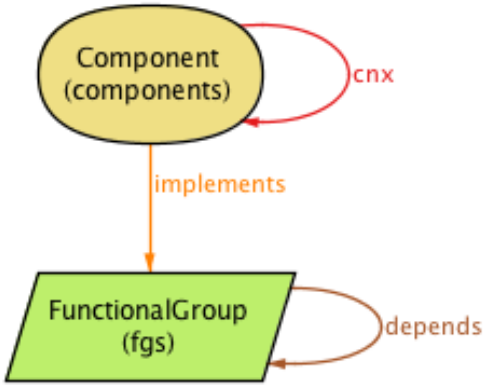


Figure 3.2: Alloy metamodel for SIP model.

The only predicate worth mentioning is `SIP_Depends_Connection_Rule`, shown in Listing 3.5.

Listing 3.5 Rule for component connections in the SIP Alloy model

```
// FG depends => Component cnx
pred SIP_Depends_Connection_Rule[sip : SIPSystem]{
  sip.components <: cnx in sip.components <: implements.depends.~
     implements
  all disj fg1, fg2 : sip.fgs | fg2 in fg1.depends and DiffComponent[
     fg1,fg2]
      => ComponentConnection[fg1, fg2, sip]
}

pred DiffComponent[fg1, fg2 : FunctionalGroup]{
  implements.fg1 not = implements.fg2
}

pred ComponentConnection[fg1,fg2 : FunctionalGroup, sip : SIPSystem]{
  (implements.fg2) in (implements.fg1).cnx
}
```

This predicate simply ensures that all connections between components are a direct result of a dependence in their respective functional groups. This is achieved by forcing all regular connections between a system's components (`(sip.components <: cnx)` to belong to `implements.depends.~ implements` (relation between components whose functional groups have a dependence). Afterwards it is only a matter of stating that a dependence

from two functional groups of different components implies a connection between said components (`all disj fg1, fg2 : sip.fgs | fg2 in fg1.depends and DiffComponent[fg1,fg2] => ComponentConnection[fg1, fg2, sip]`).

3.2.3 Discussion

Due to the simplicity of this model, there is not much worth investigating here. However there are still a few things worth exploring.

One aspect explored was the transitivity of the *depends* relation since it is not desirable for transitive dependencies to be explicitly and forcefully present. This is because all dependencies have a connection counterpart and these connections are going to be present in the source code. So an artificial increase in the connection count is not wanted.

The analysis of several instances of the model shows a variety of situations though nothing quite worth discussing or exploring further. However, it does give us an increased confidence in our model and our understanding of SIP. Since the model is so simple it is fairly easy to understand. However, this type of analysis is always useful to deepen comprehension, if only by a small amount.

3.3 Unification

This section shows how to bridge both models into a single, unified model. As it turns out, this is actually quite simple. The purpose of this unification is to be able to take the complexity metric calculation feature from SIP and apply it to IIOT. In order to do this, we must build an intermediate model of some sort (called SIPI for now). This model has two requirements:

- It must be possible to build the model either from a IIOT model or the base information used to build such an IIOT model (a call graph).
- This model must contain all the information of a SIP model for the same system, so as to enable complexity calculation.

Putting both metamodels side by side, as shown in Figure 3.3, we already see a great number of similarities between them. These can be increased even further by discarding a few elements from each model. On the IIOT side, module classifications can be ignored as

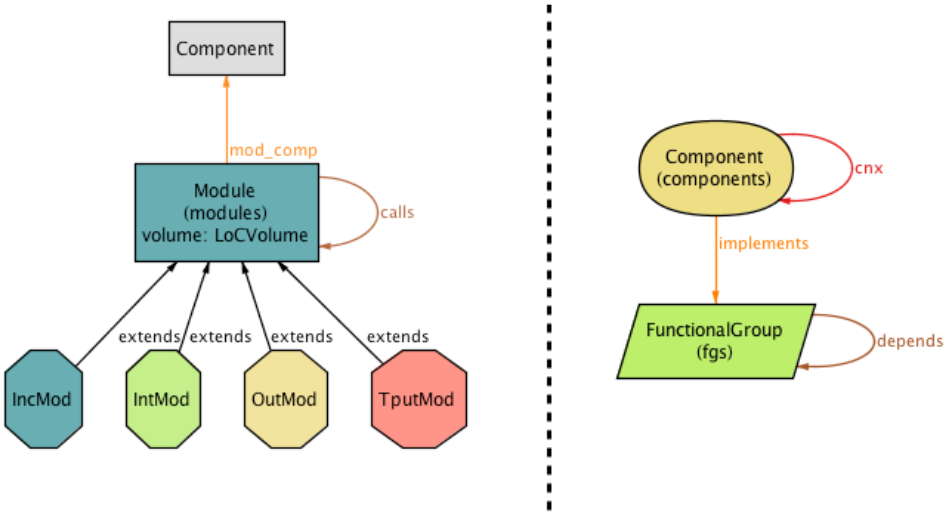


Figure 3.3: IIOT and SIP metamodels side by side.

these are not needed to calculate complexity and we can always run the classification process again. On SIP, component connections can be removed. As was already stated, connections are derived automatically from dependencies between functional groups. Therefore, complexity can be calculated by looking directly at these dependencies.

It is worth noting, however, that the only dependencies worth caring about are the ones among functional groups belonging to separate components since they are the only ones that lead to a component connection. In fact, looking at the predicate shown in Listing 3.5, we can see that every connection is either restricted by or derives from functional group dependencies.

Figure 3.4 shows both models, with the aforementioned elements hidden and their similarities become very apparent. The only real difference is the orientation of the relation between `Component` and `Module` / `FunctionalGroup` and that is mostly an Alloy detail especially since Alloy relations are bidireccional. In fact this is a very generic metamodel with containers, elements and connections between them. Both SIP and IIOT simply add a few more elements on top of this model. More, the extensions of both IIOT and SIP require no additional information: they simply make certain elements more explicit. IIOT classifies elements according to their connections and SIP adds explicit connections among components. At this point, an intermediate SIPI metamodel can be presented. It is shown in Figure 3.5 with boxes highlighting the IIOT and SIP extensions.

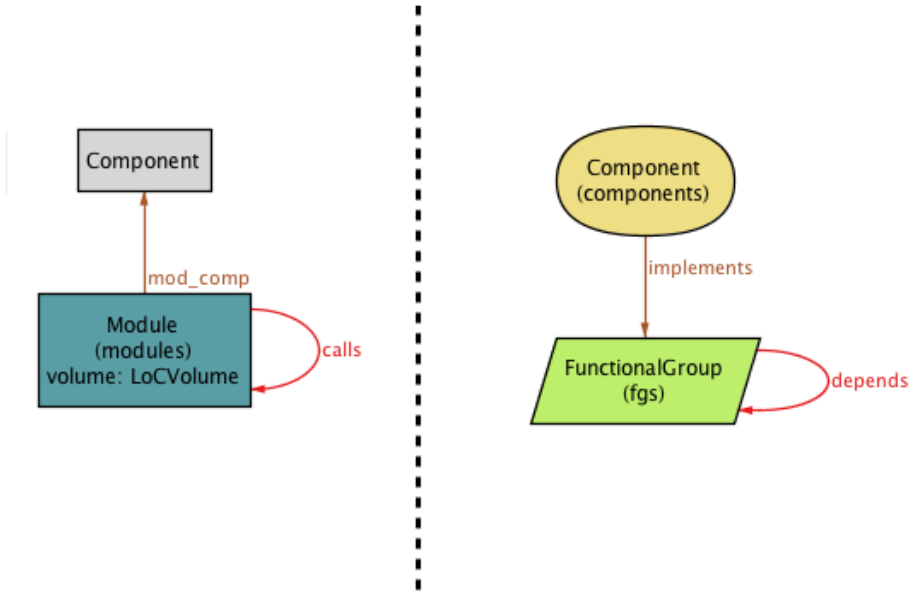


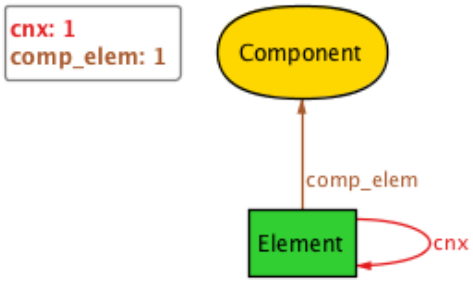
Figure 3.4: IIOT and SIP metamodels side by side w/ hidden elements.

This model provides a basis for proceeding with our work. In terms of a possible future implementation, the model can simply be enriched as needed, with extensions from SIP, IIOT or both.

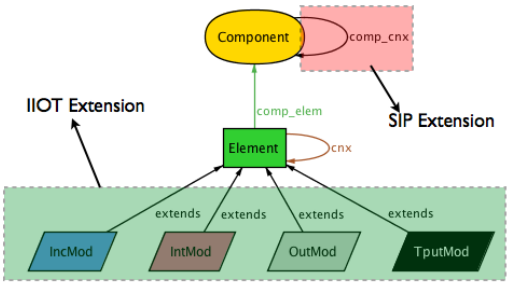
3.4 Summary

This chapter is devoted to analysing two architectural models relevant for this dissertation: IIOT and SIP. This analysis has allowed for an increased understanding of the models and served as preparation for the remaining work of this dissertation. Both models are fundamentally sound and accessible. They should pose no problems in the upcoming steps of the project.

Surprisingly, both models are intrinsically compatible with each other. This simplifies work going forward since there is little need to develop a complex intermediate representation. Indeed, either model can be used as a baseline and the other one can be represented as an extension.



(a) Basic Model



(b) Model with highlighted extensions

Figure 3.5: Unified SIP and IIOT model

Chapter 4

From Methods to Functional Groups

In order to utilise the SIP complexity metric with the IIOT model, it is necessary to work at a higher abstraction level than simply source code files. There is thus a need for somehow clustering source code units into an intermediate level of abstraction (somewhere between file and package) that would be analogous to SIP's functional groups. This chapter discusses techniques for carrying out such a clustering process. The same test case - Jpacman, a Java implementation of Pacman used at TU Delft is used throughout.

To guide the clustering experiments, several research questions were devised. Rather than focusing on the conversion from source code units to functional groups, the problem was instead tackled from a more open perspective as reflected in the research questions presented in section 1.4.

4.1 Early Approaches

The starting approach to the problem is fairly informal. The goal is to investigate various available options in order to attain the desired method groupings. Actual clustering techniques are avoided early on. The insight being that the goals are simple and small enough that they can be accomplished without resorting to sophisticated clustering algorithms.

While the idea of using clustering is revisited later on, the restriction to call graphs was definitive. The main reason being that it was an interesting problem with a sufficiently narrow scope. And since call graphs are also used for model construction, the decision to use them exclusively makes sense.

The first approach consists of utilising something we dub *micro-patterns*. A micro-pattern is in essence a specific flow of methods (successive calls in the graph). The idea is that each

Figure 4.1: Cropped method call graph of our test case system.

functional group matches a micro-pattern. By analysing the graph and grouping together the methods that match the various patterns, usable method groupings will emerge. A few examples of micro-patterns follow:

Lone Caller: A method is only called by one other method.
Rule: The called method belongs to whatever functional group the caller belongs to.

Symmetric Calls: A method calls and is called by another method.
Rule: Both methods belong to the same functional group.

Utility Calls: A method is called by various unrelated methods (no calls between them).
Rule: The method belongs to a special "Utility" functional group.

While this approach seems interesting from the outset (some work was even done on a pattern catalogue) the logistics of it soon prove too complicated. After developing a few patterns, we try to test them on a fairly simple software system. However, even relatively small systems have fairly large call graphs. The call graph for the test system used contains *404 edges and 209 nodes*. Figure 4.1 shows a cropped version of this graph and should give an idea of the complexity of the graph. Because of this, working with the graphs without significant tool support is hopeless. Manual application of the patterns is just not viable. Furthermore, most methods often fall into multiple patterns. This will obviously lead to massive overlap between functional groups. While it has not yet been settled whether or not to allow overlap in the groupings, this much is clearly excessive.

With the prospect of implementing a somewhat complex tool for pattern analysis looming ahead and considering that even preliminary experiences are yielding discouraging results, time and resources are best invested elsewhere. Especially since this area of the project is secondary.

4.2 Mathematical Approaches

The second approach to the problem is very different from micro-patterns. The focus now turns to techniques with some pre-existing tool support. We also want to use a less heuristic approach this time. Therefore, we explore some concepts from graph theory [24].

The idea is to partition a graph into subgraphs and use these as as a starting point for the grouping process. These components could be considered a sort of proto functional groups.

It is obviously not viable to look for strongly connected components because call graphs are seldom strongly connected. However, weakly connected components can be used. A weakly connected component is a subgraph whose underlying undirected graph is connected Figure 4.2). These subgraphs are much more likely to occur in a call graph. Our test case can be divided according to weakly connected components. However, the resulting breakdown consists of only two sub-components. One containing 2 nodes and another containing the remaining 207. This is a very poor starting point to build functional groups so we once again move on to another approach.

4.3 Clustering Techniques

Since all other attempts failed, it is time to apply clustering algorithms to the problem. Clustering is a very wide area of research so one needs to narrow focus. There is a lot of work done in software clustering although quite a lot of it is based on source code [49], which we are avoiding.

In the same vein, another area of research is worth considering: community detection [22], a hot subject in the field of social network research. A network (or graph) is said to have community structure if it contains sub-groups of nodes with denser connections between them than the outside nodes. There is a great deal of research on community detection in social networks, often using clustering techniques. This area of research is worth exploring to check if its results can help with the construction of method groupings. We have an intuition that there might be some crossover. After all, if one looks at a software system

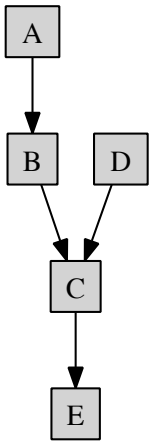


Figure 4.2: A weakly connected graph.

as a group of people (methods) communicating (calling) with each other, then it stands to reason that a community of people will be somewhat analogous to a business function. After all, if a group of methods call each other more frequently then it is likely that they are being used together somehow in the context of a shared functionality.

As there are several clustering algorithms available to choose from, our choice is governed by existing tool support. Two algorithms will be used: edge betweenness [34] and voltage [67].

The edge betweenness algorithm produces 13 clusters (see Figure 4.3). The voltage algorithm can produce any number of clusters (to some extent) since the number of clusters to be produced is a parameter of the algorithm. While interesting, this moves away from an automated analysis. The number of functional groups must be known in advance in order to perform the analysis correctly. Therefore, we focus our validation efforts on the results of the edge betweenness algorithm.

Looking at Figure 4.3, there is already a semblance of structure present in the system, with something akin to the traditional three layers of a software system (presentation, control and data).

To further organise the system and connect these results to the architectural complexity metric, the notion of components is reintroduced. All methods in the system already belong to a specific component. A method belongs to whatever component its class (or file) belongs

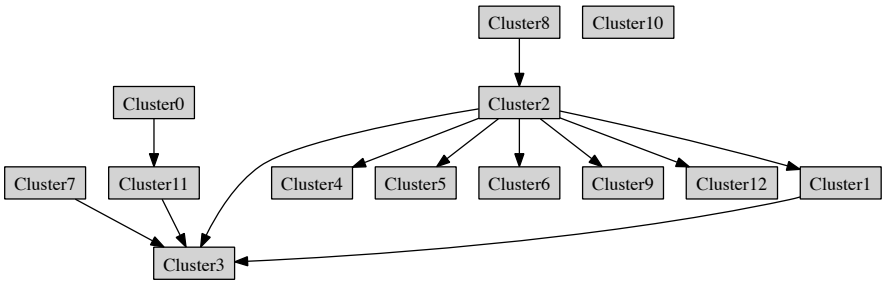


Figure 4.3: The clustered graph produced by the edge betweenness algorithm.

to. It is not important how exactly one arrives at these components (whether by manual analysis, looking at Java packages, etc.). What is important is combining the clustering results with component information. Each cluster is placed in the component to which its methods belong. If a cluster has methods from multiple components it is simply separated and dependencies reflecting this are introduced. The results of this process are shown in Figure 4.4.

Looking at Figure 4.4, we see that two clusters have been separated: cluster 2 and cluster 3. This is no coincidence as they are the two largest clusters. Nonetheless, it does resemble a structured system. But how can one be sure that these groupings actually match up in some way with the functional groups of the system? Especially without knowing what these functional groups are. By performing manual inspection on the various clusters produced by the edge betweenness algorithm one can at the very least have an idea of what kind of functionality each cluster may hold. This is done mostly by going through the names of the various methods and guessing their functionality. Obviously this is a somewhat speculative analysis.

The clusters and their respective functionalities are as follows:

Cluster 0: 4 methods all related to image handling.

Cluster 1: 1 method which displays the user interface.

Cluster 2: 158 methods with no real unifying function. Some kind of miscellaneous utility group perhaps.

Cluster 3: 26 methods relating to the control of the player character.

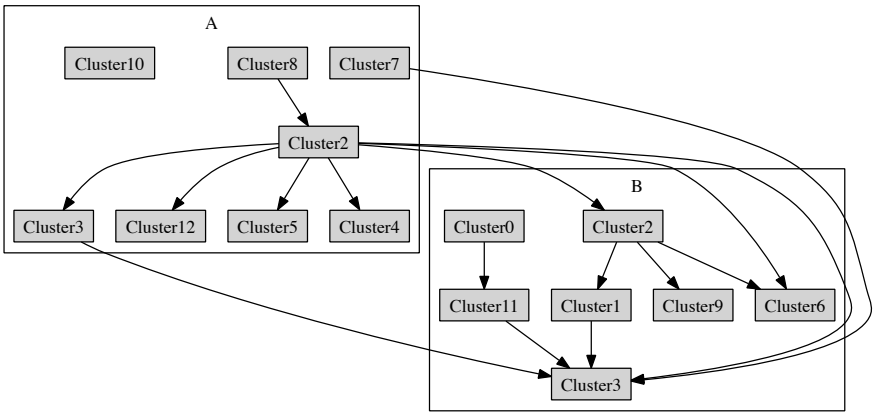


Figure 4.4: Clustered graph with components.

Cluster 4: 2 methods relating to the movement of the enemies.

Cluster 5: 1 method which converts the game board to a string.

Cluster 6: 2 methods related to updating game status.

Cluster 7: 2 methods related to the game engine.

Cluster 8: 6 methods related to the loading of various game assets.

Cluster 9: 1 method which checks for actions performed.

Cluster 10: 2 methods which perform checks on invariants.

Cluster 11: 2 methods related to constructing an image of the game board.

Cluster 12: 2 methods related to the movement of the player character.

There are obvious problems with these results, notably cluster 2. While each other cluster seems to be representing a somewhat different functionality, there is still some confusion. It seems the problem is that the algorithm stops with a large number of methods still unclustered and these are placed in a single cluster (2).

Even though manual inspection is not very reliable it nonetheless seems to indicate that the clusters produced are not very adequate. Additional analysis is needed to confirm this.

This time the clustering results are compared with those from a tool called Featureous[1]. This tool allows one to analyse a Java system and identify its features [55] (which are somewhat similar to functional groups). Featureous is a dynamic analysis tool (it requires executions of the program) and is not automated. The user must take an active role in identifying the features mostly by annotating entry points in the source code. What Featureous does is associate each method with a feature.

The Featureous analysis contains 8 features. Four of these are related to character movement and the others are related to game flow control. This points to an immediate problem in that there are significant differences between a feature and a functional group. Features map directly to user actions whereas functional groups are more varied (multiple groups may be responsible for a single feature or one group may be responsible for several features). In order to combat this, it is decided to factorize the Featureous groups since multiple methods show up in more than one group. After this operation, there are 12 Featureous groups which matches better with the existing clusters. We must further analyse and compare the results from Features and clustering[2].

The analysis reveals a problem. Several methods do not belong to any feature. This means that Featureous could not capture them or they are simply not used by the system. Either way, this complicates the analysis greatly. But even if these methods are removed, there is still little to no relevant overlap between the clusters and the features. This only serves to reinforce the idea that the groupings produced by clustering do not generally match up with the system's functional groups.

4.4 Final Remarks

Several approaches to clustering call graphs in order to produce functional groups were explored. Earlier attempts produced no viable results. Clustering techniques from the field of community detection were indeed able to produce groups. However, analysis of the outcome revealed that such groups do not really line up with our expectations of what a functional group would be.

It is possible (and likely) that a call graph simply does not contain sufficient information from which to derive the functional groups of a system. In order to construct meaningful functional groups of methods one possibly needs the source code itself.

[1] http://featureous.org/

[2] This was done with coloured diagrams. Because they are coloured, they were relegated to the appendices.

In terms of how these results affect the work going forward, they do so very little. Functional groups are not actually needed to compute our metric so the work can proceed. However, we find that the best approach to the overall problem is to make clustering yet another parameter of the process. The initial data can be enriched with clustering information from any number of sources and then combined with either method or component information in order to compute variations on the metric. Exactly which clustering technique (or if any) to use can simply be left up to users. They need only apply the technique (possibly even through another tool) and then simply enrich the existing data with new information regarding the cluster each method belongs to. One may even apply various different clustering techniques, computing and comparing the metric for each of them.

After studying both models, we realised they were already sufficiently compatible for our purposes. Therefore, even though we could successfully go up in abstraction layers on the IIOT model (in order to bring it closer to SIP) we can forge ahead in our work.

Chapter 5

Applying the SIP Metric to IIOT Models

In any project, it is always important to check and analyse the previous work. This chapter explains ground work laid for that analysis. It explores the architectural complexity metric in greater detail and presents a prototype tool (named IIOTool) to automate the analysis process.

5.1 A Formula for Complexity

In order to gain some additional insight into the architectural complexity metric being used, we present the steps originally taken by Roger Sessions to arrive at a formula for complexity [60]. Sessions began working from Glass's Law [35][1]:

> For every 25 percent increase in problem complexity, there is a 100 percent increase in solution complexity.

By applying Glass's Law to SOAs, we arrive at two notions:

- a 25 percent increase in business functionality[2] of a service leads to a 100 percent increase in the complexity of such a service;
- a 25 percent increase in the number of (external) connections of a service leads to a 100 percent increase in the complexity of such a service;

[1] The law is not stated outright although it can be inferred.

[2] Please note that *business functionality* is an equivalent term to *functional group*, presented in a previous chapter.

In order to build a formula to measure complexity one starts with a basic system. Let us consider a hypothetical system with a complexity of 1 unit. This is a system with no connections and only one functionality. This is the simplest possible system, according to Glass's Law. Complexity increases as new business functionalities and connections are added. Therefore, for a system with fg functionalities and cx connections, complexity (C) is calculated by Bird's Formula[3] (a mathematical formulation of Glass's Law), as follows:

$$C = 10^{\frac{\log(2)\times\log(fg)}{\log(1.25)}} + 10^{\frac{\log(2)\times\log(cx)}{\log(1.25)}} \tag{5.1}$$

This formula can be simplified by replacing and combining the constants.

$$\begin{aligned} C &= 10^{\frac{0.30103\times\log(fg)}{0.09691}} + 10^{\frac{0.30103\times\log(cx)}{0.09691}} \\ &= 10^{\overbrace{3.10}^{\text{Glass's Constant}}\log(fg)} + 10^{3.10\log(cx)} \end{aligned} \tag{5.2}$$

Equation 5.2 gives the complexity of a single system. Most systems, however, tend to be made up of several sub-systems which we call components. By adding up the complexity for all and components one attains the complexity value for the system as whole. Therefore, the complexity of a system with m components is given by Equation 5.3, Session's Formula for Complexity:

$$\sum_{i=1}^{m} 10^{3.10\log(fg_i)} + 10^{3.10\log(cx_i)} \tag{5.3}$$

where fg_i and cx_i are, respectively, the number of functionalities (or functional groups) and number of connections in component i.

5.2 Discussing the Metric

With a formula for complexity available, we can compute the metric for any system we have modelled. In order to pave the way for automated analysis, the metric must be integrated with the unified model. The metric presented works naturally with SIP. Since the unified model contains all the information of the regular SIP model, it should be easy to use it to compute the metric. But let us take the examination further. The complexity metric works with three

[3]The values of 1.25 and 2 are related to the 25% and 100% increases in functionality and complexity. No rationale was given for the usage of logarithms.

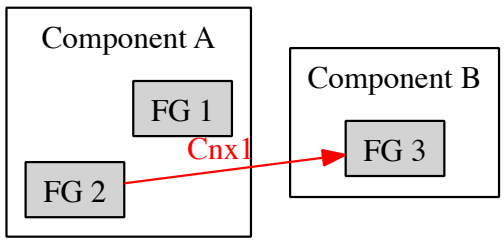


Figure 5.1: A system with two components and one connection.

elements: business functions, components and connections. We therefore need to establish parallels between these and the entities of the unified model, this being quite simple:

- business functions are Elements in our model (or Modules in IIOT);
- components are the same in both cases;
- connections only matter between components. So we ignore any connections in our model between elements of the same component.

Now the complexity for any system represented in the unified model can easily be computed . But what exactly is being measured? Exactly what is complexity in the context of this metric and model?

Complexity is a function of the number of modules in each component and the number of connections between components. Therefore, modules give one obvious (and highly important) aspect of complexity: size. All things being equal, a component with more modules tends to be more complex than a component with less modules. This seems adequate, on an intuitive basis.

Another aspect of complexity are the number of connections between components. It is important to note the following: since complexity is calculated on a per-component basis, each connection contributes twice towards the overall system complexity (once for each of the two components it connects). As an example, consider the system shown in Figure 5.1. The complexity for this system can be easily calculated.

$$
\begin{aligned}
C &= \sum_{i=1}^{2} 10^{3.10\log(bf_i)} + 10^{3.10\log(cx_i)} \\
&= \overbrace{10^{3.10\log(2)}}^{\text{FGs of C1}} + \overbrace{10^{3.10\log(1)}}^{\text{Cnx1}} + \overbrace{10^{3.10\log(1)}}^{\text{FG of C2}} + \overbrace{10^{3.10\log(1)}}^{\text{Cnx1}} \\
&= (...) \\
&\approx \overbrace{140.8498}^{\text{FGs of C1}} + \overbrace{1}^{\text{Cnx1}} + \overbrace{1}^{\text{FG of C2}} + \overbrace{1}^{\text{Cnx1}} \\
&\approx 143.8498
\end{aligned}
$$

As can be seen, the one connection in the system contributes to the complexity value twice. This is always the case.

As for how connections influence complexity, one can generally think of the connections of a system as a whole as being the “wiring” of the system. Once again, all things being equal, a system with more connections is expected to be more complex than a system with less connections. This is also intuitive and one need simply imagine a large mess of wires to get an idea of what is meant here.

This complexity metric, much like the model it is used with, can be computed at many different abstraction levels. Provided components, elements and connections are available, the complexity of system can always be computed . When comparing or analysing multiple systems it is essential to work at the same abstraction level for all of them. Generally speaking, complexity increases as one goes down in abstraction levels and decreases as one goes up. This is quite natural since, after all, abstraction is always meant to simplify something.

While we can work at many levels, we most often work at the source-code file level, the same one as IIOT works at. In this case, the elements are source code files and components are groupings of these files (for example, packages in the Java language). At this level, the values obtained from the complexity formula are often very high. Therefore, to simplify matters we will apply a logarithmic scale as in the case of, for example, the Richter scale [10]. We will use a base 10 logarithmic scale. This also has a side benefit of giving a very basic scale of complexity. For every increase of 1 unit, a system increases in complexity by one order of magnitude. Thus, the final formula will be:

$$
C = \log(\sum_{i=1}^{m} 10^{3.10\log(fg_i)} + 10^{3.10\log(cx_i)}) \tag{5.4}
$$

5.3 IIOTool Functionalities

Now that we understand the architectural complexity metric, we can develop a prototype tool to assist in the analysis of the metric. In this section we present the key functionalities of this prototype. Here follow the requirements set up for such a tool:

Reading Call Graphs The tool must be capable of reading calls graph and related information.

Constructing IIOT Models The tool must be able to construct an internal representation of the IIOT model for the system under analysis.

Calculating Complexity The tool must be able to compute the complexity metric for a system after having read it and built its IIOT model.

Visualizing Information The tool must be able to produce a visual representation of the IIOT model and it must also communicate the complexity metric value of the system in some way.

These are very simple requirements. However, it should be noted that the sole purpose of this tool is to enable testing of our working methodology. There is no concern with developing a fully-featured industrial-strength tool. We simply need something that will allow us to quickly test the work on a variety of systems so as to validate it.

5.4 Working with Call Graphs

Call graph extraction is a well-established field and there are already tools that can perform this work, for example GNU cflow [4]. In this dissertation, there is no concern with this step, as call graphs are supplied by SIG.

The call graphs are represented in comma-separated values (CSV) files. Each file contains a column indicating the caller element, the called element and the number of calls. As a matter of convenience, call graphs for three levels of abstraction are provided: method, file and component. A short example is provided in Table 5.1.

[4] http://www.gnu.org/s/cflow/

fromMethod	toMethod	Calls
NamespaceHandler.init()	ConfigBeanDefinitionParser.$constructor()	1
NamespaceHandler.init()	BigBeanDefinitionParser.$constructor()	1
NamespaceHandlerResolver.resolve(java.lang.String)	NamespaceHandler.init()	1

Table 5.1: An excerpt of a call graph in CSV form.

Component	File
beans	/src/main/java/org/springframework/beans/factory/xml/NamespaceHandler.java
aspects	/src/main/java/org/springframework/aop/config/ConfigBeanDefinitionParser.java

Table 5.2: An excerpt of component information in CSV form.

File	LoC
/src/main/java/org/springframework/beans/factory/xml/NamespaceHandler.java	10
/src/main/java/org/springframework/aop/config/ConfigBeanDefinitionParser.java	367

Table 5.3: An excerpt of file size information in CSV form.

It is worth noting that the information present in the call graph is insufficient for the purposes of our model. We also need component information. This information is also provided by SIG, again in the form of a CSV file with a *(component, file)* pair per line as shown in Table 5.2. Finally, file size information if also used and is made available to us in a CSV file with a *(file, LoC)* pair per line as shown as shown in Table 5.3.

The call graph information must be stored and unified. Since the initial call graph and component information are supplied in CSV files it was decided to continue using this format for the rest of the process. The format is simple, cross-platform and suffices for our purposes.

We will attempt to combine and store information according to some basic data warehouse principles [17]. Specifically, we will store all the disparate information (component rules and multiple levels of calls) in a single (CSV) file. Normalisation restrictions will be relaxed on this file.

The data warehouse will be composed of a CSV file where each row will store multiple pairs of *(caller, callee)* as well as the number of calls at the lowest abstraction level. In other words, the file will contain $2n + 1$ columns, where n is the number of abstraction levels.

However, it is worth noting that in order to build the model, minimum of 2 abstraction levels is needed. The CSV file has the following generic naming scheme:

fromX the origin of the call for this particular abstraction level. We can store as many as necessary although we need a minimum of 2 for the IIOT model.

toX the destination of the call. Must always match up with the *from* columns.

calls the total number of calls, always referring to the smallest abstraction level pair in the *from/to* columns.

There are obviously many different configuration such a file might take. A sample configuration along is presented along with a few rows in Table 5.4.

fromComponent	fromFile	toComponent	toFile	Calls
util	StringUtils.java	aop	ClassFilter.java	7
aop	ClassFilter.java	aop	ClassFilter.java	4
aspect	AbstractAspectJAdvicer.java	util	NumUtils.java	12
util	NumUtils.java	aspect	AbstractAspectJAdvicer.java	2
aspect	AbstractAspectJAdvicer.java	aop	TruePointcut.java	5

Table 5.4: A few rows from a data warehouse storing method, class and component call information.

Typically, the structure will be much than the one shown in Table 5.4 that merely contains 5 columns for the sake of brevity and space. Throughout our work the following 9 elements will usually be utilised: *from/to* Component; *from/to* File; *from/to* FLoC (file volume); *from/to* Method; Number of method level calls.

The first step of the process will usually be the construction of this data warehouse, containing the multiple *(from, to)* pairs. As further information is added (for example, from clustering), new pairs of columns can simply be added in (such as *fromCluster*, *toCluster* or *fromFG*, *toFG*). There are functional dependencies between the *(from, to)* pairs at lower levels and the paris at more abstract levels (for example, between file level and component level) though they will not be checked by the tool.

When building the models we will not need the information present in all the columns at once, so it is simply a matter of constructing projections on the appropriate columns.

Specifically, when constructing the model, we will usually need information on components, modules (files), volume and call counts. Different versions of these columns can be instantiated though the most typical one will involve components and files. Obviously, due to this approach we will have duplicate information at some levels.

For example, say our data warehouse has call graph information for the method, class and component levels. If we wish to construct an IIOT model for classes and components, we will need to perform a roll-up[5] of the necessary information. OLAP research has produced many interesting and efficient solutions (both commercial and academic) to handle this kind of situation [64]. An approach based on relational algebra has also been proposed "in house" in [47].

The specifics of the approach are not particularly important at this point. In fact, due to the moderate size of the call graphs utilised, there was never any real need for such high speed techniques. However, should this eventually become a problem, it would be easily solved. So it is always worthwhile to make our techniques compatible with pre-existing work.

5.5 Prototype Development Notes

In this section we will give some light notes about the development process behind the tool. This will be brief since the tool was merely a proof of concept prototype.

The tool was developed in Java. The tool has a need for high portability across various systems, and Java amply provides that. Also, the high amount of third party libraries and packages existing in Java are quite useful. As we will deal with CSV files, graphs and the like, pre-existing packages will help working with them.

In terms of handling input files, we were indeed able to use a third party package (OpenCSV[6]) that takes care of everything, giving us the information in lists of arrays. From thereupon, manipulating the information is quite simple.

The construction of the IIOT model itself was the most interesting part of the process. We were fortunate to have some knowledge of the IIOT algorithm itself though we cannot discuss it here since it is confidential information. However, the algorithm posed no real problems.

To implement the model's data structures, we simply consider the component to be an-

[5] In case of our data warehouse, roll-ups will be summations of the calls at lower levels. For example summing calls at the file level to obtain call information at the component level.

[6] http://opencsv.sourceforge.net/

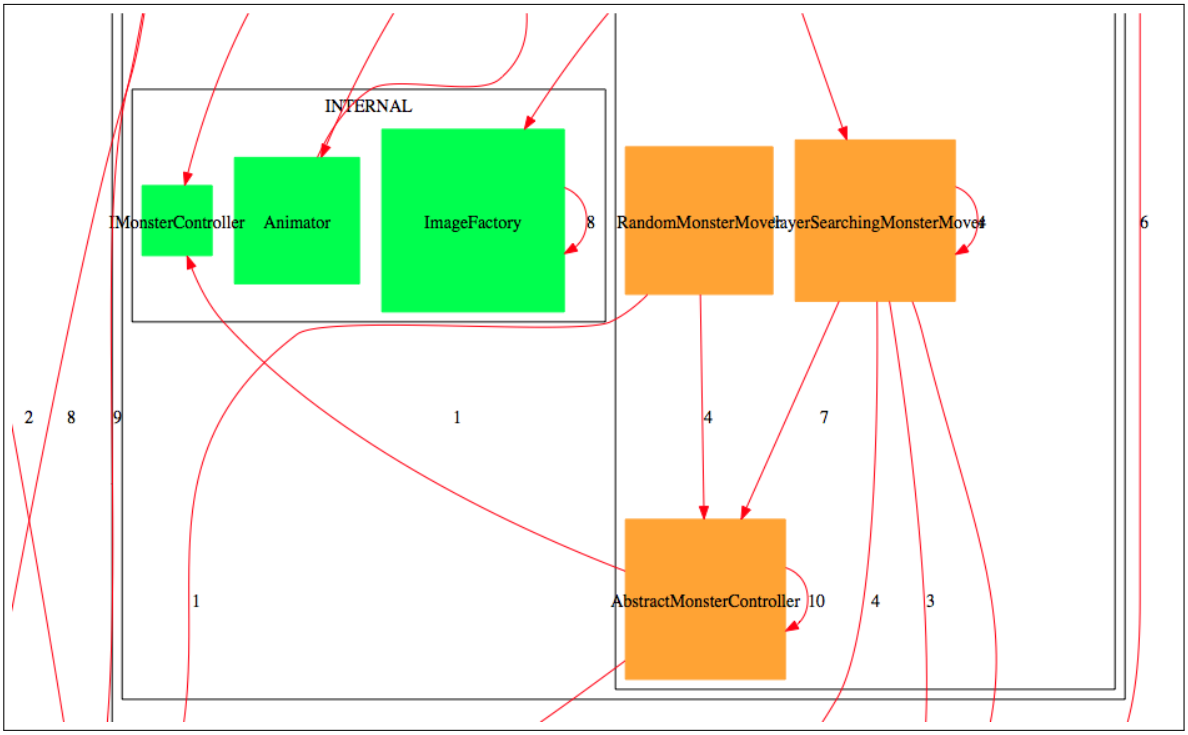


Figure 5.2: Crop of an example IIOT model constructed with DOT.

other characteristic of a module. In this case, an IIOT model is simply a kind of graph. We then implement it as such by resorting to a dedicated Java package for graph functionality[7].

In terms of producing visual outputs of the models, we decided to use DOT for graph specification [8] since it supports everything needed, visually differentiating the various module types and grouping them by component as well as providing a visual representation of the component - a “box”. The tool only produces an IIOT model’s DOT file. It is then necessary to use the DOT application itself to produce a graphic (such as a pdf file). Since DOT is an open source program that is not a problem. An example of such an output is shown in Figure 5.2. Please note that due to size constraints we have cropped the image somewhat.

5.6 Usage

In this section we will give a brief explanation on how to use the tool.

IIOTool works via command line and is packed in a .jar file. It takes system information

[7] http://jung.sourceforge.net/
[8] http://www.graphviz.org/

in the form of .csv files and produces a DOT version of the IIOT diagram. It also outputs the system complexity to the console.

IIOTool can read the information from two kinds of .csv files:

- a single, data-warehouse style file;
- 3 separate files containing call, volume and component information;

Usage is as follows:

- For single file: java -jar IIOTool.jar [csv file] [output filename]
- For multiple files: java -jar IIOTool.jar [calls file] [volumes file] [components file] [output filename]

The .csv files must have the following format:

- First line contains headers.
- For single file style, the columns must be: ‘‘fromComponent’’,‘‘fromFile’’,‘‘toComponent’’, ‘‘toFile’’,‘‘calls’’
- For multi file style, the columns must be:

  calls example: ‘‘fromModule’’,‘‘toModule’’,‘‘calls’’

  volume example: ‘‘file’’,‘‘loc volume’’

  components: ‘‘component’’,‘‘file’’

The tool will output the system complexity to the console and generate a DOT file of the respective IIOT model. To build a pdf from the DOT file, one need simply use the DOT command (eg: `dot -O -Tpdf [filename]`).

The existence of the three file input version has a simple reason behind it. The three file version was developed because it is particularly suited to connect with SIG's toolset. The toolset already exports the call, component and volume information in three .csv files. The tool takes care of combining these files into a single list of arrays and then proceeds normally. In the regular case of the single file, the list of arrays is read directly from the input file.

When using the single-file approach some care must be taken with column selection. It is possible and even likely to have more columns of information in the data warehouse than the ones required. In this case a selection must be made. Two levels of abstraction are always.

Component and file are the traditional ones for IIOT but you can use any other pairs you like. Additionally the call counts must also be supplied. Selection of these columns must be done manually (perhaps with a spreadsheet application). Either way, the file supplied to the tool must have the correct columns. Selection could have been easily added as an option to the tool but since it was not necessary for our tests, we chose to leave it out.

When selecting columns from the data warehouse .csv file, there will likely be a need for a roll up on the call counts since they always refer to the lowest level of abstraction. As a matter of convenience, the tool already takes care of this. The files can be submitted with repeated pairs and the tool will add up the call counts before proceeding. However, as previously mentioned, should the need ever arise, this step can be done ahead of time with a dedicated, high performance tool. In this case, the tool would simply bypass its own roll up step.

5.7 Future Improvements

As mentioned above, the tool is developed as a mere prototype to serve as proof of concept and by allowing us to conduct tests on a variety of systems so as to properly analyse our metric. Because of this, it is quite simple and rudimentary. However, throughout the development process we naturally came upon a few interesting ideas for improvements and new features if the objective ever went beyond a simple prototype and into a modern, fully-featured tool. We will discuss some of such ideas next.

The biggest improvement would be in terms of interfacing once we add a modern GUI. There are some advantages to command line usage, namely it is quite easy to use bash scripts to set up multiple executions and measure several systems quickly. However, for detailed analysis and exploration of a single system, a graphical interface is the best option.

This graphical tool would load up the initial CSV file(s) into a database and allow for a series of customisable visualisations from there. It would display the various abstraction levels and columns and allow the user to select which ones to use when generating an IIOT model.

Another possible level of customisation would be concerned with clustering. The system would allow the user to select both the data as well as the clustering algorithm. The resulting clusters would then be added to the existing data warehouse.

The other major benefit of such a tool would be in terms of model exploration. Ideally, we would be able to see the model and rearrange and resize it as desired. But the truly

interesting part would be exploring each node. One could begin by simply showing a graph of the various components. Each component could then be “unpacked” and the tool would display the subgraph of such component’s modules. A module could also be opened to display its classes or methods. The tool would allow one to open or close any element thus navigating across the various abstraction levels. One could even attach the source code of a file and visualise that if desired. Filtering options (for example, not displaying any internal modules) would also be a useful addition. Obviously, the tool should allow one to export the current visualisation.

These are just a few ideas that might constitute the basis for a more evolved tool that could be used to support analysis of a software system based on the work presented here.

5.8 Summary

We have presented and explored in depth a metric for SA complexity. The metric presented measures complexity as a function of the number of functional groups present in each component and the number of connections between components. The metric is computed on a per component basis and by adding the individual results for all components we obtain the overall system complexity value.

We have also presented a basic prototype tool that can be used to construct an IIOT model and use it to compute the architectural complexity metric given an appropriate set of inputs: call, component and file volume information in the form of CSV files. In the next chapter we will use this tool to perform validation on our work.

Chapter 6

Studying Architectural Complexity

Now that we have a working metric and a prototype tool to compute it, we can perform some serious testing and analysis. This chapter will present about the tests and evaluation performed.

6.1 Preparation Work

To evaluate our work we have available a series of systems to test. We compute SIP's architectural complexity metric for a number of systems and analyse the results. Let us first describe the systems under analysis.

We use 62 systems from SIG's repository of systems. Most of these are commercial, closed source systems. There were however 19 open source systems including Ant [1], JUnit [2], Tomcat[3], GlassFish [4] and others. We obviously cannot disclose information about the proprietary systems. However we can say that they come from a broad variety of areas such as finance (banking, insurance, etc.), the railroad industry, logistics, e-commerce, public administration and IT. Such a wide variety of systems provided for a solid analysis.

In addition to testing the metric across several systems, we test its evolution across time (actually, across multiple iterations of the same system). To do this, we choose two systems from SIG's repository and analyse multiple snapshots of these systems. The systems and snapshots are as follows:

[1] http://ant.apache.org/
[2] http://www.junit.org/
[3] http://tomcat.apache.org/
[4] http://glassfish.java.net/

System A: 57 snapshots spreading between 2006-10-09 and 2011-09-10.

System B: 45 snapshots spreading between 2008-08-25 and 2011-04-20.

In addition to calculating SIP's architectural complexity metric, we compute several additional metrics on these systems. There are two reasons for this. On the one hand, we want to be able to compare SIP's metric to other metrics and see what kinds of relation exist among them. On the other hand, we can use some of these metrics to further characterise the various systems under analysis.

The metrics collected are the following:

Architectural complexity the metric that has been presented before in this dissertation.

Volume the volume of the system, measured in LoC.

Component balance (CB) a metric that classifies the decomposition of a system into components developed by SIG [11]. It varies between 0 (worst balance) and 1 (best balance).

Component count the number of components in a system.

Dependency count the number of dependencies between components in a system.

6.2 Questions and Hypotheses

As we prepare to execute our tests, there is a series of objectives of analysis and a series of questions and hypotheses that need to be answered. They are described below.

Objective 1 Analyse Metric by itself.

- Evaluate the variation of the metric across several systems. See if it follows a statistical distribution.

Objective 2 Compare the architectural complexity metric with other metrics.

- Compare the architectural complexity metric and component balance. A weak correlation is expected because the metrics measure different aspects of a system.

- Compare number of dependencies and architectural complexity. A strong positive correlation is expected.
- Compare architectural complexity and volume. The expectation is that higher volumes lead to higher complexity values.

6.3 Running the Tests

Executing the tests themselves is fairly straightforward. The architectural complexity metric is collected through executions of the prototype tool presented in chapter 5. Multiple executions of the tool are chained together and coordinated via shell scripting. The results themselves are stored in simple test files.

The other metrics are collected through analysis performed by SIG's toolkit. The results for each system are then exported and stored in a CSV file.

Since there are no performance concerns, there is no need to create or calibrate a specific testing environment. All tests are executed on a standard machine.

The information for all metrics spread across the various files is then combined into a single CSV file (one for the 62 systems and one for each of the multi-snapshot systems). These files are then used as input to a statistical analysis tool: R[5].

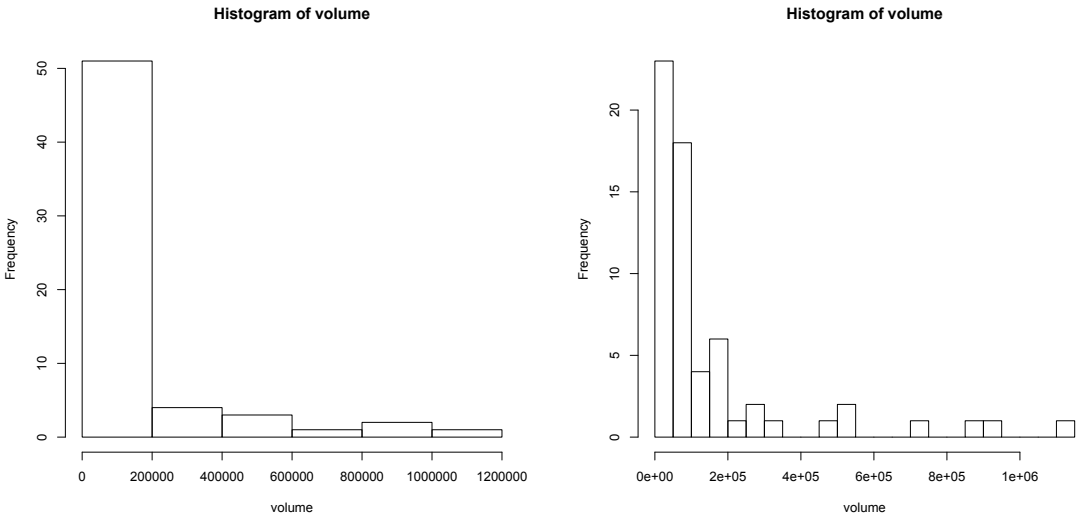
6.4 Results for multiple system

In this section we present the main set of results of our analysis: the metrics across 62 systems and corresponding statistical analysis.

6.4.1 Statistics on Metrics

The first step of our analysis is to compute a series of descriptive statistics on the various metrics in order to have a better understanding of characteristics of the systems under test. These results are presented in Table 6.1.

[5] http://glassfish.java.net/



(a) Volume histogram (coarser)

(b) Volume histogram (finer)

Figure 6.1: Volume distribution across multiple systems.

Metric	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Volume	2190	26850	65370	152700	168300	1111000
CB	0	0.1328	0.2480	0.2527	0.3602	0.6819
Component Count	1	6	9	13.1900	12.7500	81
Dependency Count	1	13.2500	24	48.4000	52	334
Complexity	3.9980	7.5720	8.7970	8.7420	9.7100	12.4300

Table 6.1: Statistical analysis for metrics across multiple systems

6.4.2 Analysis

Looking at the various metrics (architectural complexity excepted), we begin to get an idea of the distributions of these metrics in this dataset. Looking at volume we see that there is great variety, with the smallest system having approximately 2 kLoC and the largest having 1,111 kLoC. When looking at the remaining statistics we see that most of the systems involve dozens of thousands LoC. To further illustrate this point, we refer to the histogram showing the volume distribution of the various systems (Figure 6.1).

Looking at the other metrics, they only further reinforce how varied the systems under test are. Their distributions can be consulted in Section C.1 of the appendices. It should be

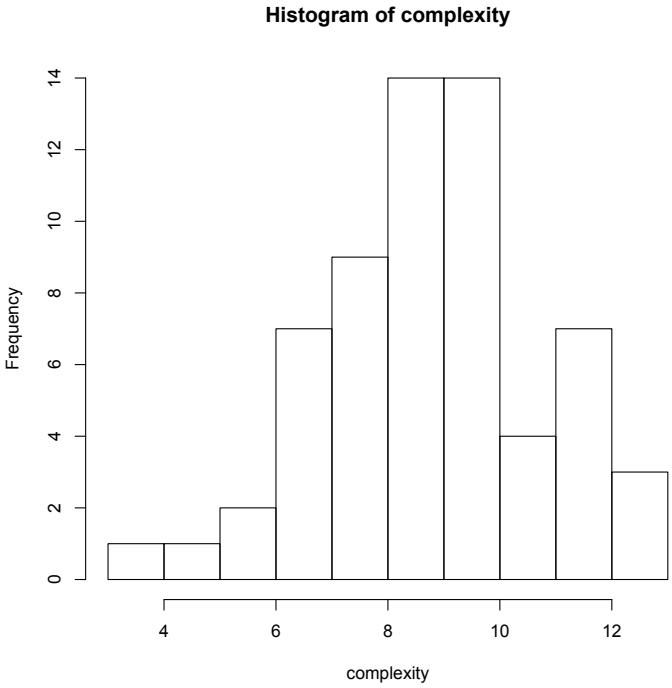


Figure 6.2: Architectural complexity distribution across multiple systems.

kept in mind that the CB metric, by definition, only varies between 0 and 1.

Turning our attention to SIP's architectural complexity metric, we see that the lowest complexity value is 3.9980 and the highest is 12.4300. Since the metric is on a logarithmic scale, this means there are close to 8 orders of magnitude of complexity in our systems. The mean and median values for complexity are both approximately 8.7.

Moving on to the distribution of the architectural complexity metric, as shown in the histogram in Figure 6.2, there is some indication that the metric might conform to a normal distribution (notice how the histogram somewhat matches the bell-curved shape of a normal distribution). To further investigate this possibility, we perform normality tests [62][6] as summarised in Table 6.2. All tests yield the same result. We cannot reject the null hypothesis in any of them. Therefore we cannot rule out that the architectural complexity metric may indeed follow a normal distribution. To further interpret this data, we use a Q-Q plot[7], as shown in Figure 6.3. When looking at the plot, the linearity of the points strongly suggests that complexity does indeed follow a normal distribution.

Test	**Variable**	**p-value**
Cramer-von Mises	W = 0.038	0.7165
Anderson-Darling	A = 0.288	0.6074

Table 6.2: Normality tests for architectural complexity distribution across multiple systems

6.4.3 Correlations on multiple systems

We now focus on the relations among the various metrics, especially how architectural complexity relates to other metrics. This will allow us to know if, for example, complexity can be used as an indicator for metrics or which, if any, of the other metrics influence the complexity score of a system. Although we focus on the relations between complexity and the other metrics, we also look at relations across some of the other metrics themselves.

In order to assess correlations among the various metrics, we calculate Spearman's rank correlation coefficient [50] (ρ) between various pairs, as summarised in Table 6.3.

[6]While a single test might suffice, we perform two for thoroughness' sake since they weigh the dataset differently.

[7]A Q-Q Plot plots the quantiles of two probability distributions against each other.

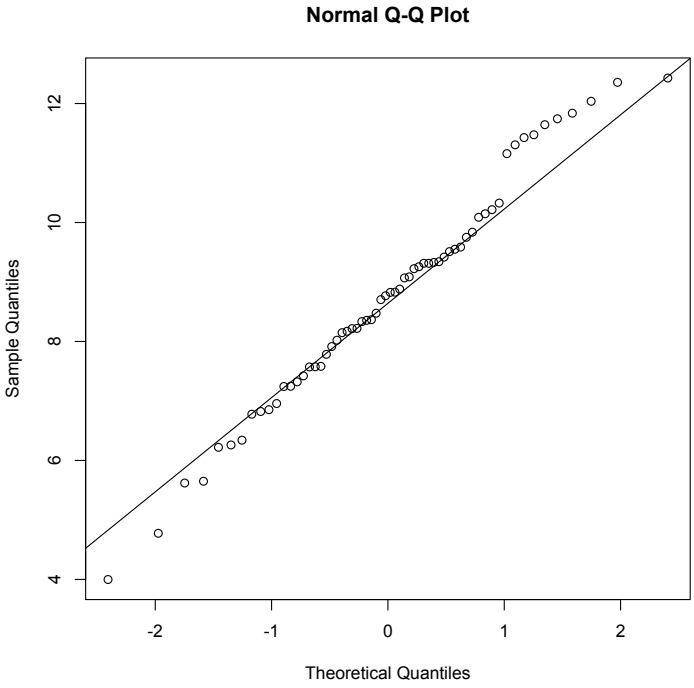


Figure 6.3: Q-Q plot for architectural complexity distribution across multiple systems.

Metrics	ρ	p-value
Volume-CB	-0.1858479	0.1481
Dependencies-CB	-0.1153027	0.3722
Components-CB	-0.2753522	0.03031
Components-Dependencies	0.9064642	< 2.2e-16
Volume-Components	0.5036307	3.01e-05
Volume-Dependencies	0.5477675	4.07e-06
CB-Complexity	-0.1059648	0.4124
Dependencies-Complexity	0.4431262	0.00031
Components-Complexity	0.3724213	0.002875
Volume-Complexity	0.91695	< 2.2e-16

Table 6.3: Metric correlations across multiple systems

6.4.4 Analysis

Looking at Table 6.3, there are quite a few pairs where the null hypothesis cannot be rejected due to the p-values surpassing our significance level (0.05). For these pairs, it is not possible to conclude anything. We therefore focus on the remaining pairs.

Looking at CB, volume and component and dependency count we see that there is weak correlation among them. CB is mostly independent from those three metrics and we can evaluate its relation to architectural complexity on its own.

When looking at component and dependency count, we see a very strong (positive) correlation between them ($\rho \approx 0.906$ with a p-value of approx. 0.0000). This is something we were already expecting and makes intuitive sense. The more components a system has, the more potential for a high amount of dependencies in that system. And on the other hand, a system with a high number of dependencies has to have a high number of components.

If we look at the relations between volume and components/dependencies we see that there is some (positive) correlation between volume and component and dependency count ($\rho \approx 0.5057$ and $\rho \approx 0.548$ respectively). In the case of components, this once again makes intuitive sense. As the number of components in a system increases, the system probably increases in terms of LoC (or vice-versa).

Let us now look at the dependence between the architectural complexity metric and the various other metrics. When looking at architectural complexity and CB, we see that the

correlation between them is very weak, as anticipated. It is possible that while both metrics measure architectural quality they do so in very different ways. CB focuses on the balance between relative size of components whereas complexity focuses more on the "wiring" among components. This can be a good thing since we can use both metrics together to evaluate a system from different angles.

As for the dependence between architectural complexity and components/dependencies, there is indeed some (positive) correlation (ρ of approximately 0.372 for components and 0.4431 for dependencies) between the two metrics. These results align with our original hypothesis though we were perhaps expecting a stronger correlation. A possible explanation for this is that the complexity metric does not simply care about the raw number of dependencies but rather how they spread across the various components. Furthermore, multiple connections between any two components are counted separately for complexity although they only represent a single dependency. The complexity metric is also influenced significantly by the number of files in each component and this is not reflected at all in the component count. In fact, one might argue that a higher number of components probably leads to a smaller number of files in each component. Perhaps this is another reason for the relatively weak correlation between the metrics: the high number of components (and by extension, dependencies), the lower complexity values for each component and the system overall, thereby weakening the overall correlation between the variables.

Finally let us analyse the dependence between architectural complexity and volume. The correlation between these two metrics is very strong, with $\rho \approx 0.917$ and p-value ≈ 0.0000. As systems volume and its complexity are in a highly monotone relationship. As system volume increases its complexity tends to (almost always) increase. While one would certainly expect a strong correlation between these two variables, one would perhaps expect something not quite so strong.

To further help us investigate the very strong correlation between architectural complexity and volume we decided to break our data set into intervals according to volume and investigate correlation between the variables at each level. This might identify some levels as having particularly strong correlation and that might be skewing the overall values somewhat. Results are summarised in Table 6.4.

Volume Range	ρ	p-value	datapoints
0-100k	0.8271777	< 2.2e-16	41
100k-500k	0.5928571	0.02227	15
500k-1000k	0.7	0.2333	5
1000k+	Insuficient datapoints		1

Table 6.4: Stratified correlations between volume and complexity

If we look at the data in Table 6.4 we indeed see that correlation is stronger among the smaller systems (volume < 100 kLoC). Eventually we start to obtain p-values that indicate there might not be any correlation. For the highest group (> 1000 kLoC) we have an insufficient number of systems to analyse dependence.

It is worth noting that our data set is heavily weighted towards the first subgroup. As we move down in the groups we find ourselves with smaller and smaller data sets. This might obviously influence the results and this should be kept in mind.

Nonetheless we can say that correlation between volume and architectural complexity does indeed seem to be stronger for systems under 100 kLoC. Perhaps a possible explanation for this is that, at these volume levels, the size of the system affects its complexity more than the amount of connections. As size reaches a certain threshold the complexity contribution of connections takes over and volume becomes a worse predictor for complexity. Nonetheless, even at higher levels of volume, the correlation is still significant.

We would also like to note that we took advantage of these groupings and also measured correlation between architectural complexity and module and component count in each range. However, p-values for these tests were such that all results were deemed non-significant. These results can nonetheless be found in section C.2 of the appendices.

6.5 Results for two systems across time

In this section we present an alternative metric analysis. Instead of evaluating the various metrics across multiple systems, we instead focus on how the various metrics and their relationships evolve over time. In other words, across multiple versions of the same system.

6.5.1 Statistics on Metrics

We begin by presenting a statistical analysis of the metrics for both systems in Table 6.5 and Table 6.6.

Metric	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Volume	370100	502400	536500	516100	543100	555600
CB	0	0	0	0.01287	0	0.1258
Component Count	1	1	13	10.19	19	21
Dependency Count	1	1	40	33.68	66	70
Complexity	9.289	9.735	11.35	10.59	11.43	11.49

Table 6.5: Statistical analysis for metrics across multiple snapshots of system A

Metric	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Volume	140400	151800	167800	171700	170800	273000
CB	0.1902	0.2364	0.2829	0.2643	0.2941	0.3169
Component Count	7	10	10	10.24	10	13
Dependency Count	15	20	20	20.53	20	25
Complexity	8.665	8.829	9.816	9.453	9.845	10.19

Table 6.6: Statistical analysis for metrics across multiple snapshots of system B

6.5.2 Analysis

This time around the table offers an overall picture of both systems which points at their evolution and might help guide our investigations down the line. Let us make note of a few points that might be of interest.

System A

- The minimum volume is around 37 kLoC and the max is around 555 kLoC. However, if one looks at the median and quartiles, it is quite obvious that this system quickly goes to 500 kLoC and stays there.
- Component balance was never a concern for this system as it scores 0 on most snapshots.

- The fact that there is a massive leap in component count from the first to second quartiles (likewise for dependencies) is worth further investigation.
- Architectural complexity grows two orders of magnitude from minimum to maximum.

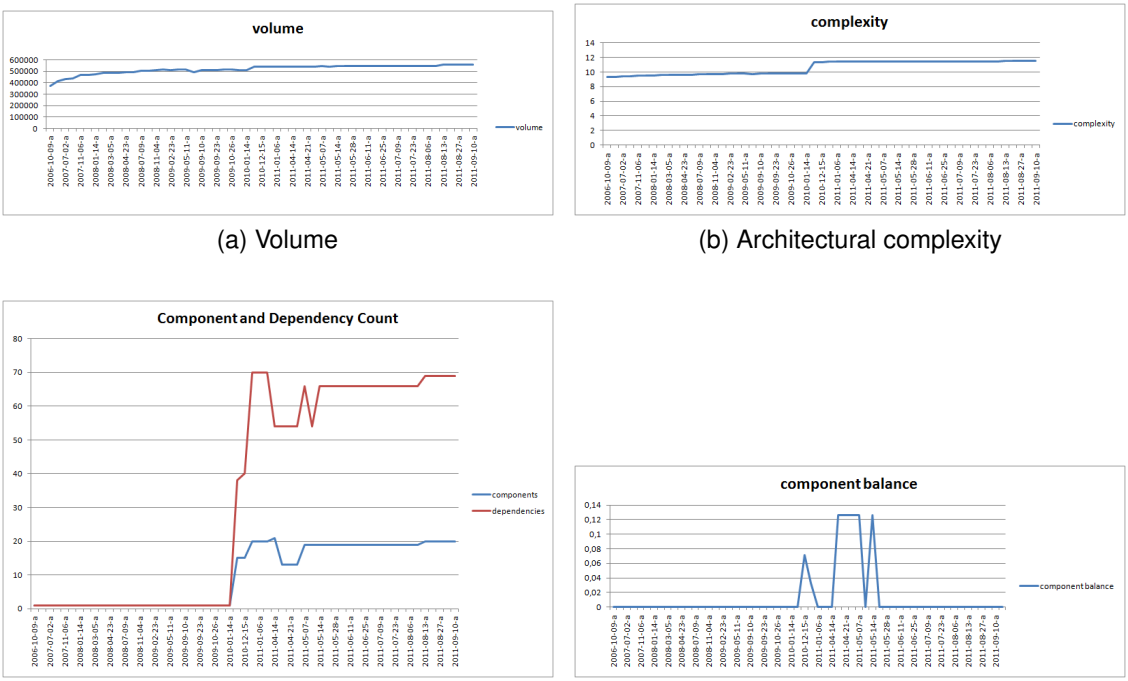
System B

- The minimum volume (140 kLoC) is around half the max (273 kLoC); however, unlike system A, the volume seems to stay under 200 kLoC for most snapshots.
- This system seems to have taken some care in balancing. In fact the score seems to be somewhat constant so the same care was probably taken throughout most of the process (ie, no single iteration dedicated to improving balance).
- The number of components and dependencies is somewhat stable for this system. This again reinforces that whatever care was taken with respect to the structure of the system, it stayed constant throughout the process.
- There is once again a difference of (close to) two orders of magnitude between min and max architectural complexity. This might indicate a possible pattern, though two systems are insufficient to make statements with any level of certainty.

These statistics are not sufficient to give a good idea of the overall behaviour of the various metrics across the lifetime of either system. Even though our data is discrete, we can order it by snapshot date and then plot the various metrics[8].

Beginning with system A, if we look at Figure 6.4 we can see a few interesting phenomena. The volume of the system grows somewhat continuously. It grows a lot faster in the earlier stages (development at this point was probably very focused on adding new features) and then slows but still continues to grow. Components and dependencies spike suddenly. However, their CB score is almost always zero (save a couple of instances that were probably "accidental") which seems to indicate that components never really change. It is possible the CB metric is unable to properly assess the structure of this system. On the other hand, looking at the architectural complexity metric, its growth is mostly continuous though there is a massive spike (two orders of magnitude) in the middle. This matches the spike in component and dependency count. Looking at all the information together it seems likely that system A went from a single monolith (one component) to having the code spread

[8]Larger images of all the plots in this section as well as combined versions can be found in section C.1 of the appendices.



(a) Volume

(b) Architectural complexity

(c) Components and Dependencies

(d) Component balance

Figure 6.4: Various metrics plotted across snapshots of system A.

across various components. This is confirmed in the data: the snapshot from 2010-01-14 has a single component with a volume of 511 kLoC and the following snapshot (2010-09-14) has 15 components with a combined volume of 536 kLoC. This introduced a great number of additional dependencies (the change is from 1 to 38 in such snapshots) and increased architectural complexity in system A. From a balance perspective it would seem they went from one extreme (monolith) to the other (excessive partitioning). It is interesting to note that SIP's complexity metric punishes the second situation a lot more.

Let us now look at system B, whose metrics are plotted and shown in Figure 6.5. The volume of system B is steadily decreasing and there is a significant dive early on. A possible explanation is that developers may have spent a few iterations explicitly trying to decrease volume. The number of components and dependencies stays constant throughout the entire process, save for a few occasional iterations. This might suggest a very regimented approach to the structuring of system B. Interestingly enough, the CB metric seems to be a bit more scattered all over the place when compared to the other two. There is a significant dip at some point though it does not match up with anything in the other metrics. Do note however that the variation of CB is between roughly 0.2 and 0.3. So in truth, one can say that CB is

(a) Volume

(b) Architectural complexity

(c) Components and Dependencies

(d) Component balance

Figure 6.5: Various metrics plotted across snapshots of system B.

constantly shifting around a stable value. The architectural complexity metric, as expected, decreases throughout most snapshots (though not without some occasional spikes). It is particularly interesting that there is a massive dip a bit after the halfway mark. Interestingly, this does not match with any dip in the other metrics. A possible explantation is that there was a significant reduction in the connections between components of system B (as defined by the architectural complexity metric). It is worth noting that this phenomenon is something none of the other metrics can capture.

6.5.3 Correlations across snapshots

We finish the analysis by looking at correlation data across the snapshots of systems A and B. Studying the dependence between various metrics across the lifetime of a software system is particularly interesting since it can help us identify which metrics can be used as predictors of each other. Let us look at the correlation data, summarised in Tables 6.7 and 6.8.

Metrics	ρ	P-Value
Volume-CB	0.04399599	0.7452
Dependencies-CB	0.05057809	0.7087
Components-CB	0.1511333	0.2618
Components-Dependencies	0.9697184	< 2.2e-16
Volume-Components	0.8631316	< 2.2e-16
Volume-Dependencies	0.884786	< 2.2e-16
CB-Complexity	0.06812918	0.6146
Dependencies-Complexity	0.8776422	< 2.2e-16
Components-Complexity	0.861033	2.20e-016
Volume-Complexity	0.9830251	< 2.2e-16

Table 6.7: Metric correlations across multiple snapshots of system A

Metrics	ρ	P-Value
Volume-CB	0.3433465	0.02142
Dependencies-CB	-0.3348231	0.02456
Components-CB	-0.2005348	0.1866
Components-Dependencies	0.9019616	< 2.2e-16
Volume-Components	0.2262697	0.135
Volume-Dependencies	0.1556441	0.3073
CB-Complexity	0.6823258	2.44e-07
Dependencies-Complexity	-0.000627701	0.9967
Components-Complexity	0.09780898	0.5227
Volume-Complexity	0.6648658	6.32e-07

Table 6.8: Metric correlations across multiple snapshots of system B

6.5.4 Analysis

Beginning with system A (Table 6.7), there are several results that we disregard due to their p-value. They all involve CB which, as previously shown, is almost constantly 0. So we cannot really conclude anything in terms of the predictive power of CB regards to architectural complexity from this system.

The correlations between volume, component and dependency counts remain very strong, which was expected. In terms of predictive power, any of the metrics can obviously be used to some extent to predict the other two.

Looking at the SIP architectural complexity metric we see that correlations between it and volume ($\rho \approx 0.983$), component count ($\rho \approx 0.861$) and dependency count ($\rho \approx 0.878$) are all strong. What this means is that, for system A at least, any of component count, dependency count or volume has some predictive power with respect to architectural complexity. And these correlations mean developers can use any of those three metrics (volume, component and dependency count) to help with attempts to reduce the complexity of system A.

Turning our attention to system B (Table 6.8), we see a somewhat different situation. The first thing we notice is that this time we must disregard the correlations between components/dependencies and other metrics. This makes perfect sense once we recall those metrics were constant throughout the entire project and the other metrics vary. Interestingly enough, we do have usable results for the correlation between dependency count and CB ($\rho \approx$ -0.335 and p-value = 0.0246). This can be attributed to the fact that though CB is not constant it does not vary too much (for example, its median absolute deviation is approximately 0.029).

If we look at CB we see that there is a somewhat strong correlation ($\rho \approx 0.682$) between it and architectural complexity. This means that, in the case of system B, CB can be used as a reasonable indicator for the complexity of the system. We have theorised previously that the developers of system B might have been very strict when structuring the system. It is an interesting side-effect that this approach led to a higher influence of structure on complexity. In other words, since the number of dependencies and components never changed, the way the files were distributed across components suddenly became much more important. This confirms the intuition gained from analysing the architectural complexity formula that it is possible to influence complexity without ever increasing the size of a system simply by rearranging the distribution of files across components.

However, there is one thing that is quite unexpected. In the CB metric, higher values mean better decompositions [11]. However, since we have a positive correlation, a decrease of CB (worsening decompositions) actually means a decrease in architectural complexity. This is unexpected. However, it should be kept in mind that this system does not score particularly well in terms of CB. The scale of that metric is [0,1] and the best that system B scores is 0.317. It is possible that for poor scores of CB, the decomposition of the system has the opposite effect on the system's architectural complexity. This could mean system B

is somehow an outlier. However it might also mean that this strange relation might actually exist. Still, it would have been interesting to actually perform a manual inspection of system B to further investigate the matter.

Finally, we look at the dependence between volume and architectural complexity for system B. The correlation between these two metrics is somewhat strong ($\rho \approx 0.665$) though less than in the multi-system tests. This means that volume is a slightly worse predictor of complexity in the case of system B. When this is taken in conjunction with everything else we have looked at, it seems that the complexity of system B is mostly influenced by the number of connections (this is the only plausible explanation). This might be a side-effect of the rigid structuring approach. On the other hand, there is the possibility that the snapshots we have examined corresponded to a dedicated effort to simplify the system (indeed both volume and complexity steadily decrease while the rest mostly stays unchanged). Perhaps the developers were more successful in streamlining the connections of the system than they were in reducing its size.

6.6 Summary

We have presented an analysis of SIP's architectural complexity metric. We have tested the behaviour of the metric by measuring several systems. We have also tracked the evolution of the metric across a system's lifetime by measuring multiple snapshots of two separate systems.

As for the metric itself, we have found that it seems to follow a normal distribution. This means very high or very low architectural complexity values can considered abnormal. It is possible to calibrate the metric so as to define threshold values for these abnormalities. When tracking the metric across the lifetime of a system we found that in one of the cases it grew steadily and in the other it decreased. In both cases there was a difference of two orders of magnitude between highest and lowest.

The correlations among the various metrics tell us how much information the SIP architectural complexity metric can give us when in relation to the other metrics. The very strong correlations with volume across the various tests indicates that the architectural complexity metric gives us little information with regards to quality than what we can glean from volume. On the other hand, architectural complexity gives us a significant amount of additional when compared to components, dependencies and CB (with the exception of one of the systems tracked across its lifetime).

Chapter 7

Conclusion

This chapter closes this dissertation by providing an overall evaluation of the project. We will discuss what we have achieved and what was not so successful. We will also offer an assessment of the main outcome of the project: the complexity metric and its main strengths and weaknesses. This will finish with a few ideas on how this work may be extended or continued in the future.

7.1 Overall Review

SA is a broad and interesting area and one of the most important ones in software development these days. With any system being developed today that is worth anything, there must be some degree of concern with SA. On the other hand, this a field of research that is not tied up with any particular technology nor is its applicability restricted to any one field of software development and research. For all these reasons, this is a very interesting area to work in.

Working with two different models was a fairly new challenge, particularly the point of integrating them. Originally this was expected to be one of the most difficult parts of the project. But as we analysed the models, we realised they were naturally compatible. SIP took some more time to analyse since there are many variations on the model and because SIP has at its heart a process for managing complexity. But we eventually found a version of SIP that was highly compatible with our objectives. From that point on, there were no difficulties in working with either model or the unified representation.

The most challenging part of the project was the work related to clustering. Most of our efforts in this area did not pay off in terms of usable results. We were never able to produce

acceptable groupings of methods by basing ourselves exclusively on call graphs. But this does not mean that our experiments were a failure or a waste of time. In fact, considering that our main objective was to investigate whether or not it was possible to derive the functional groups automatically from the source code, we do have an answer. It is unfortunate that the answer is a negative one even though that does not make it any less valid or worthy.

Looking at our work with clustering and framing it in a proper context, we realise that it was not a complete failure. We wanted to investigate the possibility of clustering methods into business groups. Our investigation concluded that it is very difficult to do so based solely on call graphs. Should producing groups be mandatory we could certainly have chosen to use other approaches when call graphs failed (perhaps source code clustering itself).

The complexity metric itself is fairly easy to understand and use. Implementing the metric on our prototype was also simple. The challenges of developing the tool had more to do with how to implement the models and datatypes than calculating the metric. But overall, that part of the project did not present many difficulties. Never meant to be a deliverable, the tool was meant to be a proof of concept of the approach based on the two models and the complexity metric. And it was of course a means to an end. It was only thanks to the prototype that we were able to perform analysis and validate the complexity metric.

The validation process itself posed no problems. We were pleased that the tool was able to analyse such a wide number of systems. We were obviously confident that the tool was working well but it was good that it passed such a workout.

7.2 Metric assessment

We regard the complexity clearly as the most interesting outcome of the entire work reported in the dissertation. There is a real need for ways to measure and evaluate the architectural quality of a software system. Obviously, a complexity metric by itself, no matter how good, cannot give us a complete quality assessment. But it is a practical way to get an overview of quality of the entire system. The proposed metric has a few major strengths and weakness that we would like to discuss for a moment.

One aspect of the metric revealed by our analysis that can be interpreted as a weakness is the very high dependence between complexity and volume. There is a very high correlation between both metrics. Volume and complexity are almost perfect monotone functions of one another. There is nothing wrong with this on a fundamental level, as it makes perfect sense that larger systems will be more complex than smaller ones. However, from a practical

point of view one might ask what is the point of using the complexity metric for system analysis when we can simply use volume and draw similar conclusions and make similar comparisons.

There are some caveats worth mentioning in this regard though. First and foremost, volume and complexity are two different things. The fact that the variations in one are matched by the other does not mean the metrics measure the same. Nor does it mean that it is impossible to have systems of small volume and high complexity or vice-versa. It is simply unlikely that this will normally occur. Secondly, we can measure complexity in the design stages. At any point where we have components and elements that go inside said components (be these files or functionalities or something else) we can compute complexity. It is impossible to compute a system's volume in the design stages. Furthermore, in the design stage it is easy to move things around. By constantly tracking the complexity of the design one can get a sense of whether or not a design change is beneficial and we also know exactly how much each change (or addition) will cost in terms of complexity.

When measuring implemented systems, there are also several reasons in favour of the use of both metrics. First of all, the complexity metric is not particularly hard or expensive to compute and therefore it can easily be computed alongside other metrics. Secondly, it is another dimension along which we may monitor a system. One can use the metric to establish several rules for managing the complexity of a system. For example, the maximum percentage that a single component may contribute towards overall system complexity or what percentage of complexity may be contributed by connections and by files. The fact that the metric works on a logarithmic scale also gives an immediate sense of the increasing of the order of magnitude of a system's complexity. The smaller numbers also make discussion around metric scores more manageable and easier to follow.

One of the metric's strengths is its simplicity, since it can be computed easily. However on the other hand the metric is built on a solid mathematical foundation. This means that it is precise, accurate and logical. There is no guess work or interpretation involved. When speaking to the various stakeholders of the system, if we discuss the system's complexity and point to the metric, it can help reach consensus by having a mathematical formula that can be agreed upon by all. Furthermore, the metric produces a single number for a system. This makes it very easy to compare various systems or various iterations of a system. Is a system getting better or worse as it evolves? Developers often have suspicions one way or the other but no objective way to be sure. By using the metric one can be certain and quantify the changes with precision.

Another strength is in what exactly the metric considers complexity. Complexity is size and relations. The fact that complexity is computed on a component basis allows us to zero in on particularly complex components. If we are monitoring various snapshots of a system and detect a sudden increase in the overall metric, we can compare the values for each component and identify exactly which ones are responsible for the spike. On the other hand, if one is attempting to refactor a system, we can use the metric to quickly identify where efforts should be concentrated in order to obtain maximum payoff. Another advantage is that, by knowing which components are the most complex in a system, we know where to prioritise maintenance and test efforts. One can even go further and evaluate files individually. By looking at the connections of each file, we can use the formula to predict the impact of moving a file from one component to another. There are many advantages to the approach. And since the values for each component are combined in the end, we lose none of the advantages of having a single overall score for the system.

7.3 Future Work

We have successfully bridged both architectural models - SIP and IIOT - and incorporated a complexity metric for SA in said models. We have also validated the metric through detailed analysis of a large case study. We also investigated the question of clustering methods into business functionalities and have concluded that it is not possible to do so by using only a system's call graph.

But none of the above means that we cannot do more. There are certainly other ideas to explore which are briefly discussed next.

For starters, there were of couple of results in our case study that might merit further investigation. We were particularly surprised with the positive correlation between the CB and complexity metrics that was observed along the lifetime of System B. A detailed manual analysis of that system might explain the phenomenon. Alternatively, it might be interesting to further explore the dependence between both metrics. Perhaps System B was an anomaly or perhaps the metrics truly interact in such a nonintuitive way. This could be investigated by studying the progression of both metrics across several other systems. And if what happened with System B repeats itself often, it would certainly be a fascinating phenomenon worth investigating.

The clustering problem we examined is still an interesting one. As we mentioned, there are other possible avenues of research. Perhaps we might attempt to combine source code

clustering with call graph clustering hoping to produce better groupings. And how would such groupings interact with our complexity metric? We could measure the complexity of the system after applying various different clustering techniques (this would be complexity more from a design or functionality standpoint than implementation/source code). And what would each of such values mean? Could the metric somehow be used to evaluate the clustering techniques? Perhaps by comparing the results to the complexity of the original system.

As for the metric itself, the next step could be to combine it with other metrics and incorporate it into a complete working quality model. We already know how the metric behaves and exactly what aspects of a system's SA it measures. Now it is simply a matter of adding it to an existing metric suite, for instance SIG's. We know that complexity is an essential aspect of quality. But perhaps we could relate complexity more directly with quality. Perhaps we could tie our metric to one or more particular quality attributes.

These are just a few possible directions where this research might be taken. There are also more practical avenues such as incorporating our tool and approach into existing tools and techniques for quality assessment. It would certainly be interesting to follow up on this work somehow as we feel there is still plenty of potential here. On the whole, this work lays down foundations for interesting future research.

Bibliography

[1] P. Abrahamsson, M.A. Babar, and P. Kruchten. Agility and architecture: Can they coexist? *Software, IEEE*, 27(2):16 –22, 2010.

[2] R. Allen and D. Garlan. Formalizing architectural connection. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 71 –80, May 1994.

[3] M.A. Babar and I. Gorton. A tool for managing software architecture knowledge. In *Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent, 2007. SHARK/ADI'07: ICSE Workshops 2007. Second Workshop on*, page 11. IEEE, 2007.

[4] M.A. Babar and I. Gorton. Software architecture review: The state of practice. *Computer, IEEE*, 42(7):26 –32, 2009.

[5] M.A. Ali Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *Proceedings of the 2004 Australian Software Engineering Conference*, ASWEC '04, pages 309–, Washington, DC, USA, 2004. IEEE Computer Society.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[7] L. Bass and B.E. John. Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66(3):187 – 197, 2003. Software architecture – Engineering quality attributes.

[8] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 308–317. IEEE, 2002.

[9] P.O. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (alma). *Journal of Systems and Software*, 69(1-2):129 – 147, 2004.

[10] D.M. Boore. The richter scale: its development and use for determining earthquake source parameters. *Tectonophysics*, 166(1-3):1–14, 1989.

[11] E. Bouwers, J.P. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Proceedings of the 9th Working IEEE/IFIP Conf. on Software Architecture. IEEE Computer Society*, 2011.

[12] E. Bouwers and A. van Deursen. A lightweight sanity check for implemented architectures. *Software, IEEE*, 27(4):44 –50, jul. 2010.

[13] E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations. In *Software Maintenance, 2011. ICSM 2011 Proceedings. IEEE International Conference on*. IEEE Computer Society, 2011.

[14] L.C. Briand, J.W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3:65–117, July 1998.

[15] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91 –121, 1999.

[16] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[17] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26:65–74, March 1997.

[18] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20:476–493, June 1994.

[19] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley, 2002.

[20] A. Cooper. *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity (2nd Edition)*. Pearson Higher Education, 2004.

[21] L. Couto, J.N. Oliveira, M. Ferreira, and E. Bouwers. Preparing for a literature survey of software architecture using formal concept analysis. In *Proceedings of the 5th International Workshop on Software Quality and Maintainability*, 2011.

[22] L. Danon, A. Diaz-Guilera, J. Duch, and A. Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005:P09008, 2005.

[23] N. Delisle et al. Formally specifying electronic instruments. In *Proceedings of the 5th international workshop on Software specification and design*, pages 242–248. ACM, 1989.

[24] R. Diestel. *Graph Theory*. Springer-Verlag, 2005.

[25] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *Software Engineering, IEEE Transactions on*, 28:638–653, July 2002.

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

[27] P.H. Feiler, D.P. Gluch, and J.J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.

[28] M. Fowler and K. Scott. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[29] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Software Engineering, International Conference on*, 1995.

[30] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is still so hard. *Software, IEEE*, 26(4):66 –69, 2009.

[31] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97. IBM Press, 1997.

[32] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1:1–40, 1993.

[33] Alan C. Gillies. *Software Quality: Theory and Management*. Chapman & Hall, Ltd., London, UK, UK, 1992.

[34] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821, 2002.

[35] R.L. Glass. *Facts and fallacies of software engineering*. Addison-Wesley Professional, 2003.

[36] A. Immonen and E. Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7:49–65, 2008.

[37] D. Jackson. *Software Abstractions: logic, language and analysis*. MIT Press (MA), 2012.

[38] S.H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[39] R. Kazman. Tool support for architecture analysis and design. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pages 94–97, New York, NY, USA, 1996. ACM.

[40] R. Kazman, L. Bass, M. Webb, and G. Abowd. SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the 16th international conference on Software engineering*, pages 81–90. IEEE Computer Society Press, 1994.

[41] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on*, pages 68 –78, August 1998.

[42] P.B. Kruchten. The 4+1 view model of architecture. *Software, IEEE*, 12(6):42 –50, November 1995.

[43] C.F.J. Lange, M.R.V. Chaudron, and J. Muskens. In practice: Uml software architecture and design description. *Software, IEEE*, 23(2):40 – 46, 2006.

[44] N. Lassing, D. Rijsenbrij, and H. van Vliet. On software architecture analysis of flexibility, Complexity of changes: Size isn't everything. In *Proceedings of the Second Nordic Software Architecture Workshop NOSA*, volume 99, pages 1103–1581, 1999.

[45] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336 –354, April 1995.

[46] C.H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97. IBM Press, 1997.

[47] H.D. Macedo and J.N. Oliveira. Do the middle letters of "OLAP" stand for Linear Algebra ("LA")? Technical report, HASLab, 2010.

[48] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering — ESEC '95*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Berlin / Heidelberg, 1995.

[49] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 45 –52, June 1998.

[50] J.H. McDonald. *Handbook of biological statistics*. Sparky House Publishing, 2009.

[51] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70 –93, January 2000.

[52] G. Molter. Integrating SAAM in domain-centric and reuse-based development processes. In *Proceedings of the 2nd Nordic Workshop on Software Architecture, Ronneby*, pages 1–10, 1999.

[53] A. Nugroho, B. Flaton, and M. Chaudron. Empirical analysis of the relation between level of detail in uml models and defect density. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 600–614. Springer Berlin / Heidelberg, 2008.

[54] L. O'Brien, P. Merson, and L. Bass. Quality attributes for service-oriented architectures. In *Systems Development in SOA Environments, 2007. SDSOA '07: ICSE Workshops 2007. International Workshop on*, May 2007.

[55] A. Olszak and B.N. Jørgensen. Featureous: A tool for feature-centric analysis of java software. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 44–45. IEEE, 2010.

[56] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, October 1992.

[57] U. Priss. Formal concept analysis in information science. *Annual review of information science and technology*, 40(1):521–543, 2006.

[58] B.G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, (3):216–226, 1979.

[59] R. Sessions. *Controlling Complexity in Enterprise Architectures - Executive Overview Part I of III*. ObjectWatch, Inc., June 2007.

[60] R. Sessions. The IT complexity crisis: Danger and opportunity. *White paper, November*, 2009.

[61] M. Shaw and P. Clements. The golden age of software architecture. *Software, IEEE*, 23(2):31 – 39, 2006.

[62] M.A. Stephens. Tests based on EDF statistics. *Goodness-of-fit Techniques*, 68:97–193, 1986.

[63] B. Tekinerdogan, H. Sozer, and M. Aksit. Software architecture reliability analysis using failure scenarios. *Journal of Systems and Software*, 81(4):558 – 575, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[64] P. Vassiliadis and T. Sellis. A survey of logical models for OLAP databases. *SIGMOD Rec.*, 28:64–69, December 1999.

[65] L.G. Williams and C.U. Smith. Pasasm: a method for the performance assessment of software architectures. In *Proceedings of the 3rd international workshop on Software and performance*, WOSP '02, pages 179–189, New York, NY, USA, 2002. ACM.

[66] K.E. Wolff. A first course in formal concept analysis. *SoftStat*, 93:429–438, 1993.

[67] F. Wu and B.A. Huberman. Finding communities in linear time: a physics approach. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):331–338, 2004.

Index of Terms

Appendix A

Alloy Models

A.1 IIOT Model

```
module IIOT

open util/ordering[LoCVolume] as locvol



//================================
// -----Sigs ----------------------------
//================================

abstract sig Module {
  volume : one LoCVolume,
  mod_comp: lone Component,
  calls: set Module
}

sig IntMod, IncMod, OutMod, TputMod extends Module {}
sig Component {}
sig LoCVolume {}
sig IIOTModel {
  modules: Module
}
```

```
//================================
// ----- Preds ----------------------------
//================================

fun fellows[mod : Module] : set Module{
  mod_comp.(mod.mod_comp)
}
// Modules belong at most to one component
pred Lone_Component [iiot: IIOTModel] {
  iiot.modules <: mod_comp.~(iiot.modules <: mod_comp) in iden
//  (iiot.comp_mod).~(iiot.comp_mod)  in iden // Relation is simple
}


// general call rules predicates
pred NoCallsOutside[mod: Module]{
  mod.calls in fellows[mod]
//  mod.(iiot.calls) in  ((iiot.comp_mod).mod).(iiot.comp_mod)
}

pred NotCalledOutside[mod: Module]{
  calls.mod in fellows[mod]
//  (iiot.calls).mod in  ((iiot.comp_mod).mod).(iiot.comp_mod)
}

pred CallsOutside[mod: Module] {
  some mod.calls
  mod.calls not in fellows[mod]
}

pred CalledOutside[mod: Module] {
  some calls.mod
  calls.mod not in fellows[mod]}


// Internal modules call rules
pred Internal_Calls[iiot: IIOTModel]{
```

```
  all intM : IntMod | NoCallsOutside[intM] and NotCalledOutside[intM]
}

// Incoming modules call rules
pred Incoming_Calls[iiot: IIOTModel]{
  all  incM : IncMod | NoCallsOutside[incM] and CalledOutside[incM]
}

// Outgoing modules call rules
pred Outgoing_Calls[iiot: IIOTModel]{
  all outM : OutMod | NotCalledOutside[outM] and CallsOutside[outM]
}

// Throughput modules call rules
pred Throughput_Calls[iiot: IIOTModel]{
  all tputM : TputMod | CalledOutside[tputM] and CallsOutside[tputM]
}


pred All_Preds[iiot: IIOTModel]{
  Lone_Component[iiot]
  Internal_Calls[iiot]
  Incoming_Calls[iiot]
  Outgoing_Calls[iiot]
  Throughput_Calls[iiot]
}



//================================
// ----- Auxiliary Checks ----------------------------
//================================


// Do all modules have a component?
pred Is_Orphan[mod : Module]{
  no mod.mod_comp
}

assert No_Orphans_A{
    all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
```

```
      no mod : calls.(iiot.modules) + (iiot.modules).calls | Is_Orphan[
          mod]
}
check No_Orphans_A for 3 but exactly 1 IIOTModel


// What happens when modules have no component?
pred Called_By_Orphan[mod: Module]{
  some caller : calls.mod | Is_Orphan[caller]
}

pred Calls_Orphan[mod: Module, iiot: IIOTModel]{
  some callee : mod.calls | Is_Orphan[callee]
}

assert No_Bad_Calls_By_Orphans{
  all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
      no cbo: calls.(iiot.modules) + (iiot.modules).calls  |
        Called_By_Orphan[cbo] and cbo in OutMod + IntMod
}
check No_Bad_Calls_By_Orphans for 6 but exactly 1 IIOTModel

assert Orphans_Call_Incoming{
  all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
      no cbo: calls.(iiot.modules) + (iiot.modules).calls  |
        Called_By_Orphan[cbo] and cbo in IncMod
}
check Orphans_Call_Incoming for 3 but exactly 1 IIOTModel

assert Orphans_Call_Throughput{
  all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
      no cbo:  calls.(iiot.modules) + (iiot.modules).calls |
        Called_By_Orphan[cbo] and cbo in  TputMod
}
check Orphans_Call_Throughput for 3 but exactly 1 IIOTModel

assert No_Bad_Calls_To_Orphans{
  all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
      no co:  calls.(iiot.modules) + (iiot.modules).calls |
        Calls_Orphan[co, iiot] and co in IncMod + IntMod
}
```

```
check No_Bad_Calls_To_Orphans for 6 but exactly 1 IIOTModel

assert Outoing_Call_Orphans {
  all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
      no co:  calls.(iiot.modules) + (iiot.modules).calls |
        Calls_Orphan[co, iiot] and co in OutMod
}
check Outoing_Call_Orphans for 3 but exactly 1 IIOTModel

assert Throughput_Call_Orphans {
  all iiot : IIOTModel | All_Preds[iiot] => All_Preds[iiot] and
      no co:  calls.(iiot.modules) + (iiot.modules).calls |
        Calls_Orphan[co, iiot] and co in TputMod
}
check Throughput_Call_Orphans for 3 but exactly 1 IIOTModel

//================================
// ----- Comms ---------------------------
//================================

// Auxiliary facts
fact Aux{
  //no IIOTModel.calls
//some OutMod
}

run All_Preds for 3 but exactly 1 IIOTModel
```

code/iiot.als

Listing A.1: Complete Alloy model for IIOT.

A.2 SIP Model

```
module SIP

//================================
// -----Sigs ---------------------------
//================================

abstract sig SIPSystem {
  fgs: set FunctionalGroup,
  components: set Component
```

```
}

sig FunctionalGroup {
  depends: set FunctionalGroup
}

sig Component{
  cnx: set Component,
  implements : set FunctionalGroup
}


//================================
// ----- Preds ----------------------------
//================================


// FG depends => Component cnx
pred SIP_Depends_Connection_Rule[sip : SIPSystem]{
  sip.components <: cnx in sip.components <: implements.depends.~
     implements
  all disj fg1, fg2 : sip.fgs | fg2 in fg1.depends and DiffComponent[fg1,
     fg2]
        => ComponentConnection[fg1, fg2, sip]
}

pred DiffComponent[fg1, fg2 : FunctionalGroup]{
  implements.fg1 not = implements.fg2
}

pred ComponentConnection[fg1,fg2 : FunctionalGroup, sip : SIPSystem]{
  (implements.fg2) in (implements.fg1).cnx
}

// cnx relation only inside SIPSystem
pred Subs_System[sip: SIPSystem]{
  (sip.components).cnx in sip.components
  cnx.(sip.components) in sip.components
}

// depends relation only inside SIPSystem
```

```
pred Deps_System[sip: SIPSystem]{
  (sip.fgs).depends in sip.fgs
  depends.(sip.fgs) in sip.fgs
}


// All FGs are implemented
pred FG_Impl[sip: SIPSystem]{
  sip.fgs = sip.components.implements
}


// Only one Component per FG
pred SIP_1Comp_Per_FG[sip : SIPSystem]{
  all fg: sip.fgs | Has_1Comp[fg]
}

pred Has_1Comp[fg : FunctionalGroup]{
  lone implements.fg
}

// Components and FGs are not subs of themselves
pred SIP_No_Self_Sub[sip: SIPSystem]{
  all c : sip.components | No_Self_Sub[c]
  all fg : sip.fgs | No_Self_Sub[fg]
}

pred No_Self_Sub[c : Component]{
  c not in c.cnx
}

pred No_Self_Sub[fg : FunctionalGroup]{
  fg not in fg.depends
}


// Depends if not two-way
pred Depends_1_Way[dpd : FunctionalGroup -> FunctionalGroup]{
  no ^dpd & iden
}
```

```
pred SIP_Depends_1_Way[sip : SIPSystem]{
  Depends_1_Way[sip.fgs <: depends]
}

// Cnx is not two-way
pred SIP_Sub_1_Way[sip :  SIPSystem]{
  Sub_1_Way[sip.components <: cnx]
}

pred Sub_1_Way[sb : Component -> Component]{
  no ^sb & iden
}




pred All_Preds[sip: SIPSystem]{
  SIP_1Comp_Per_FG[sip]
  FG_Impl[sip]
  Subs_System[sip]
  Deps_System[sip]
  SIP_No_Self_Sub[sip]
  SIP_Sub_1_Way[sip]
  SIP_Depends_1_Way[sip]
  SIP_Depends_Connection_Rule[sip]
}

//================================
// ----- Auxiliary Checks ----------------------------
//================================


// Sub Component implements FG that depends on FG implemented by
   Supercomponent
assert Sub_Impl_Depends_Super_Impl {
--  all sip : SIPSystem | All_Preds[sip] =>All_Preds[sip] and
  --  ( all comp : sip.components |
    --  no (comp.cnx.(sip.implements)).depends & comp.(sip.implements))
}
check Sub_Impl_Depends_Super_Impl for 3 but exactly 1 SIPSystem
```

```
// Dependencies inside a Component
assert Dependency_Inside_Component{
  all sip : SIPSystem | All_Preds[sip] => All_Preds[sip] and
  some disj fg1, fg2 : sip.fgs | fg1 in fg2.depends and no implements.fg1
      & implements.fg2
}
check Dependency_Inside_Component for 3 but exactly 1 SIPSystem

// 2 FGs depend on 1 and vice-versa
assert FGs_2_Depend_1{
  all sip : SIPSystem | All_Preds[sip] => All_Preds[sip] and
  all fg : sip.fgs | lone depends.fg and lone fg.depends
}
check FGs_2_Depend_1 for 3 but exactly 1 SIPSystem

// Are dependencies transitive?
assert Depends_Transitive{
    all sip : SIPSystem | All_Preds[sip] => All_Preds[sip] and
      all fg : sip.fgs | fg.depends.depends in fg.depends
}
check Depends_Transitive for 3 but exactly 1 SIPSystem

//================================
// ----- Comms ----------------------------
//================================



// Auxiliary facts to help with 'specific' executions
fact Aux{
  Component in SIPSystem.components
  FunctionalGroup in SIPSystem.fgs
  some cnx
}

run All_Preds for 3 but exactly 1 SIPSystem
```

code/sip.als

Listing A.2: Complete Alloy model for SIP.

Appendix B

Features versus Clusters

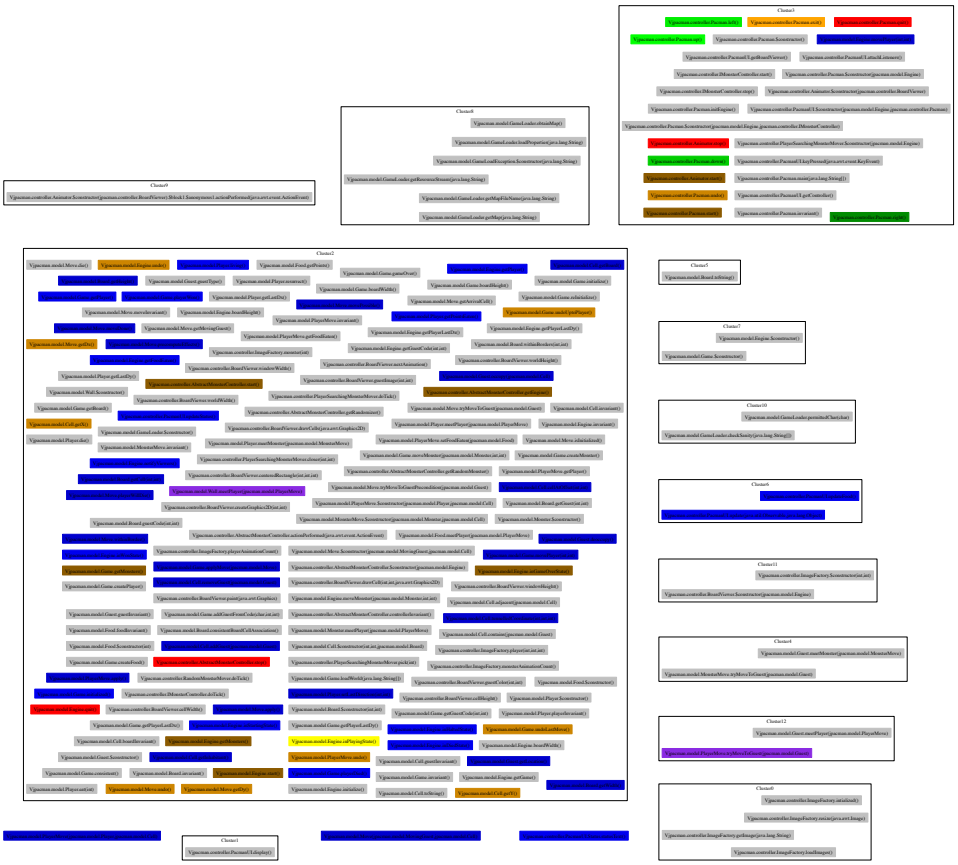


Figure B.1: Methods grouped by clusters and colored by feature group.

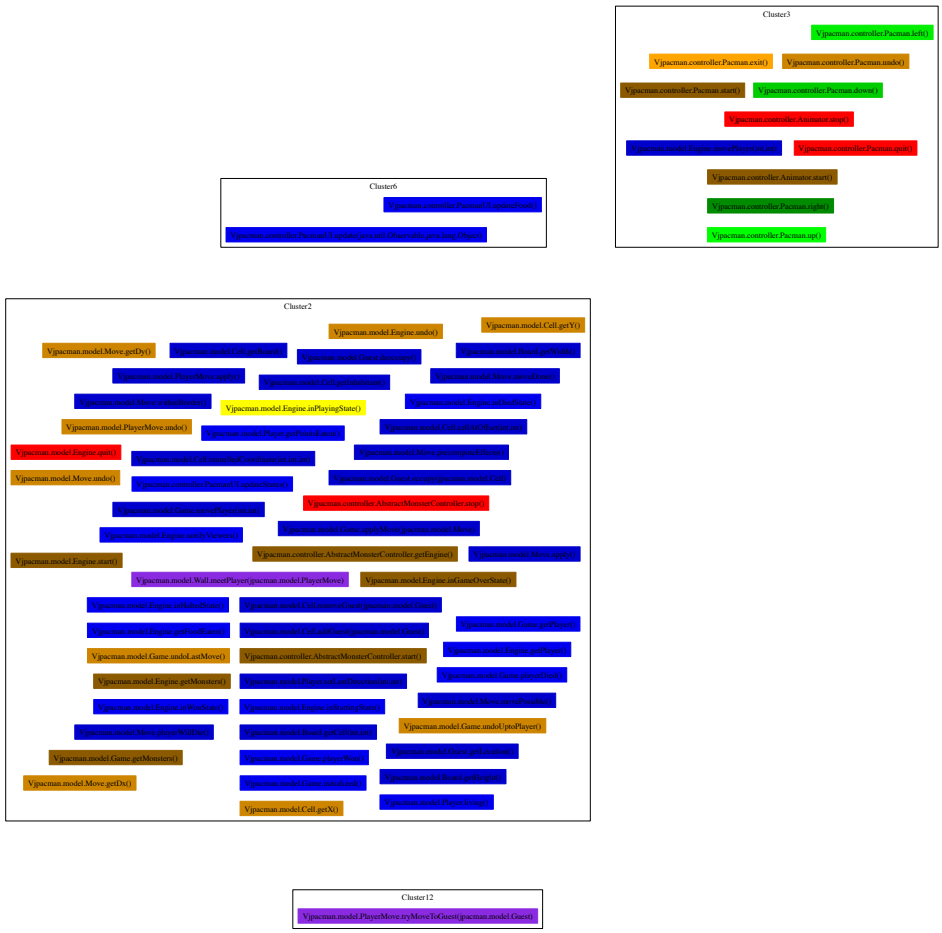


Figure B.2: Methods grouped by clusters and coloured by feature group. Methods w/o feature removed.

Appendix C

Case Studies Data

C.1 Histograms

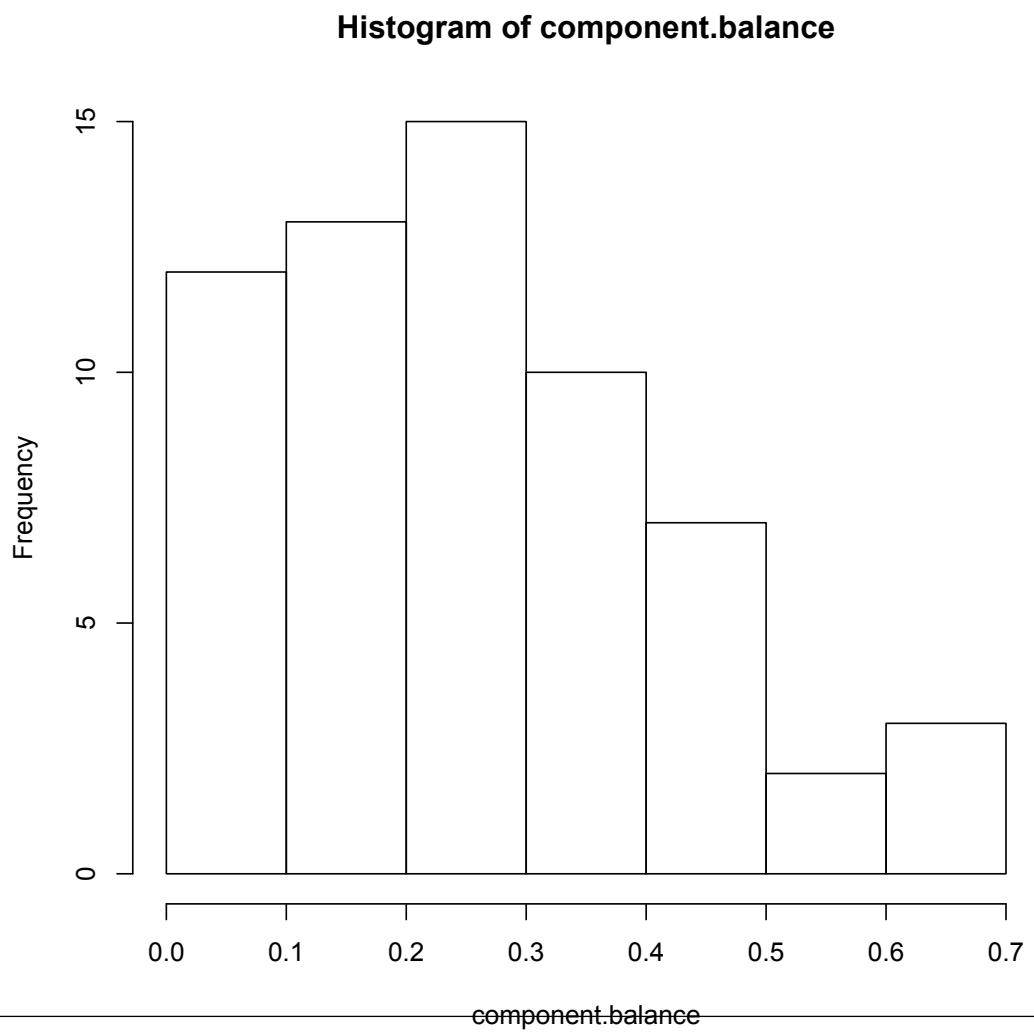


Figure C.1: CB distribution across multiple systems

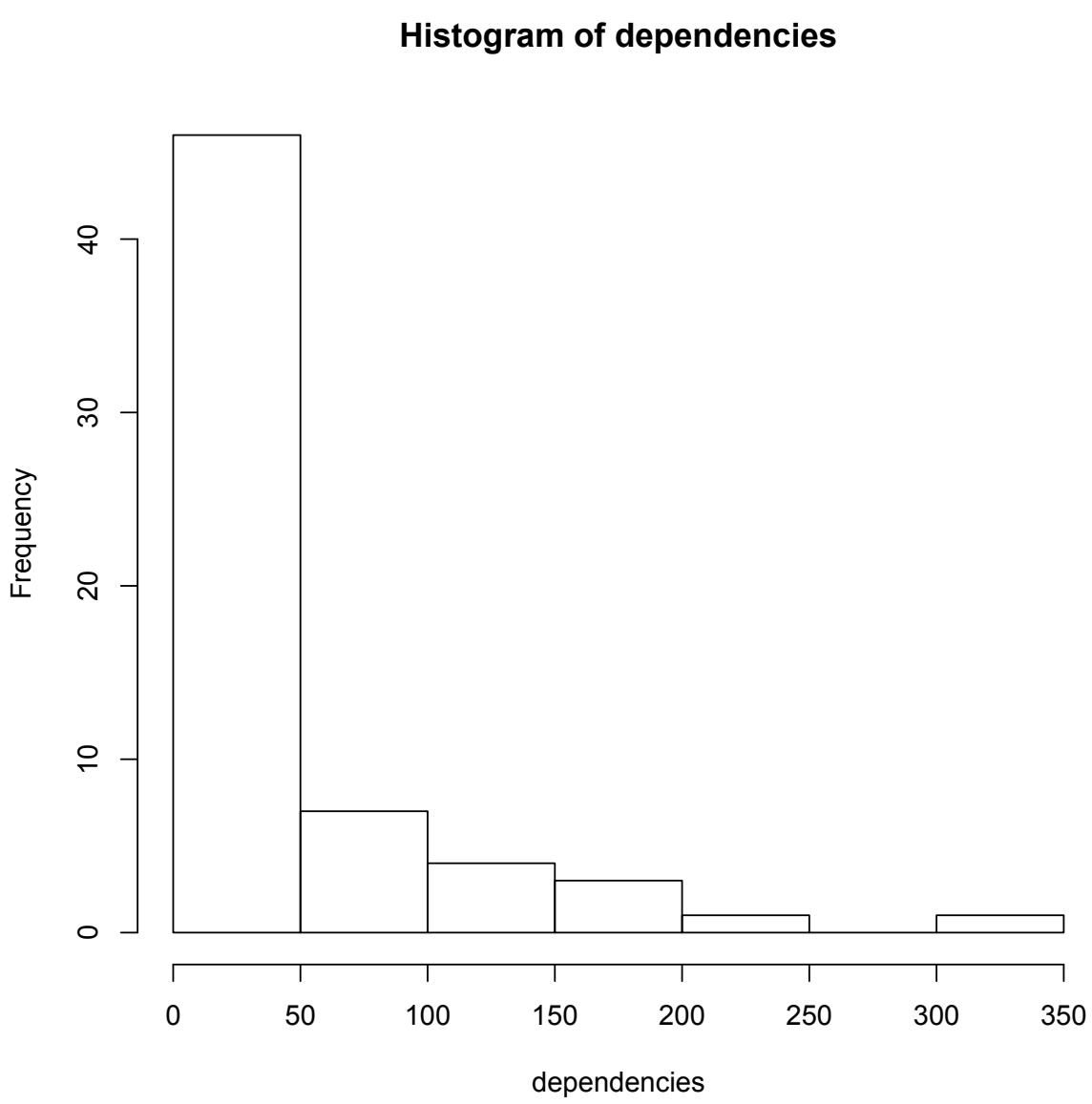


Figure C.2: Dependency count distribution across multiple systems

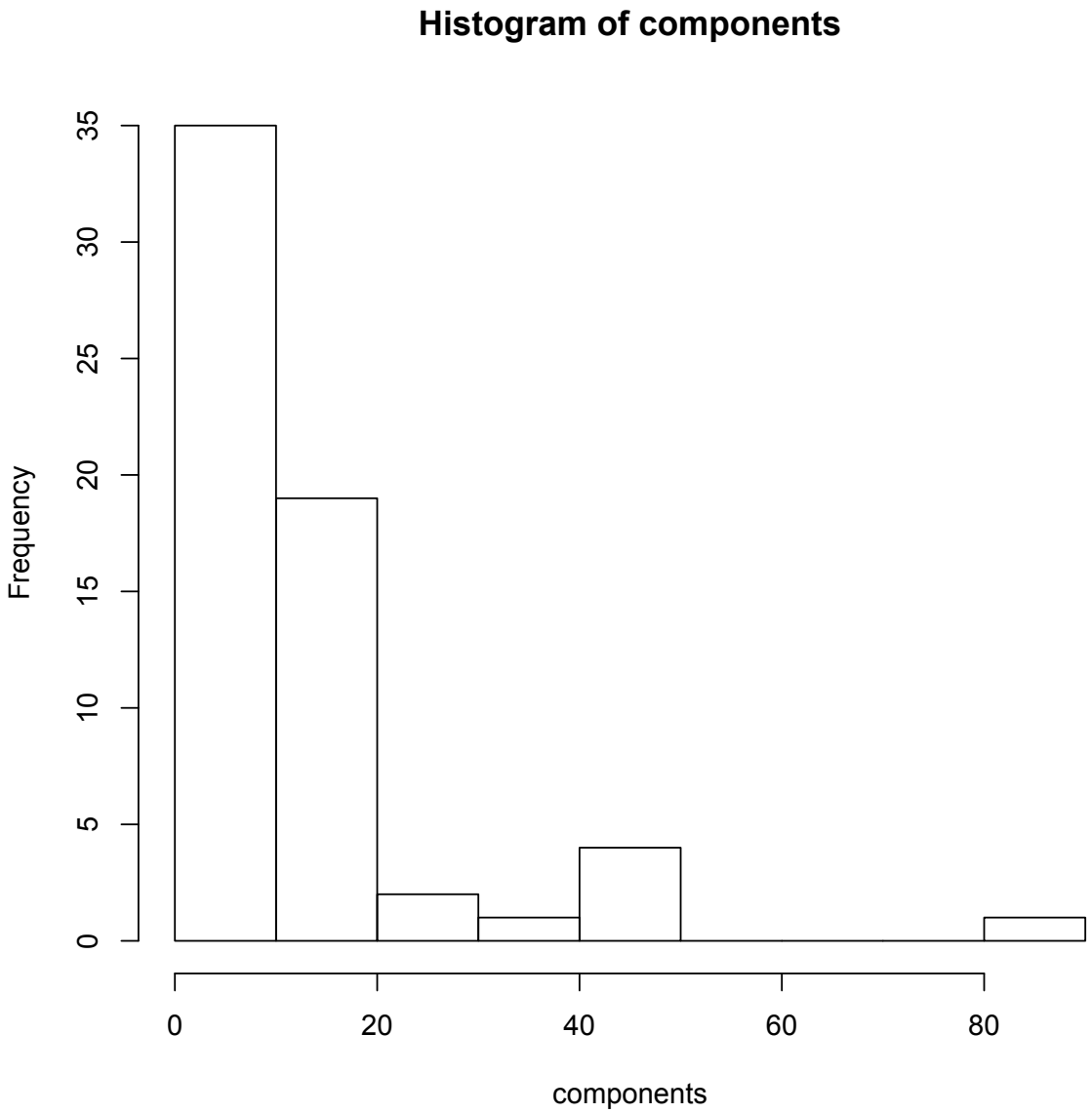


Figure C.3: Component count distribution across multiple systems

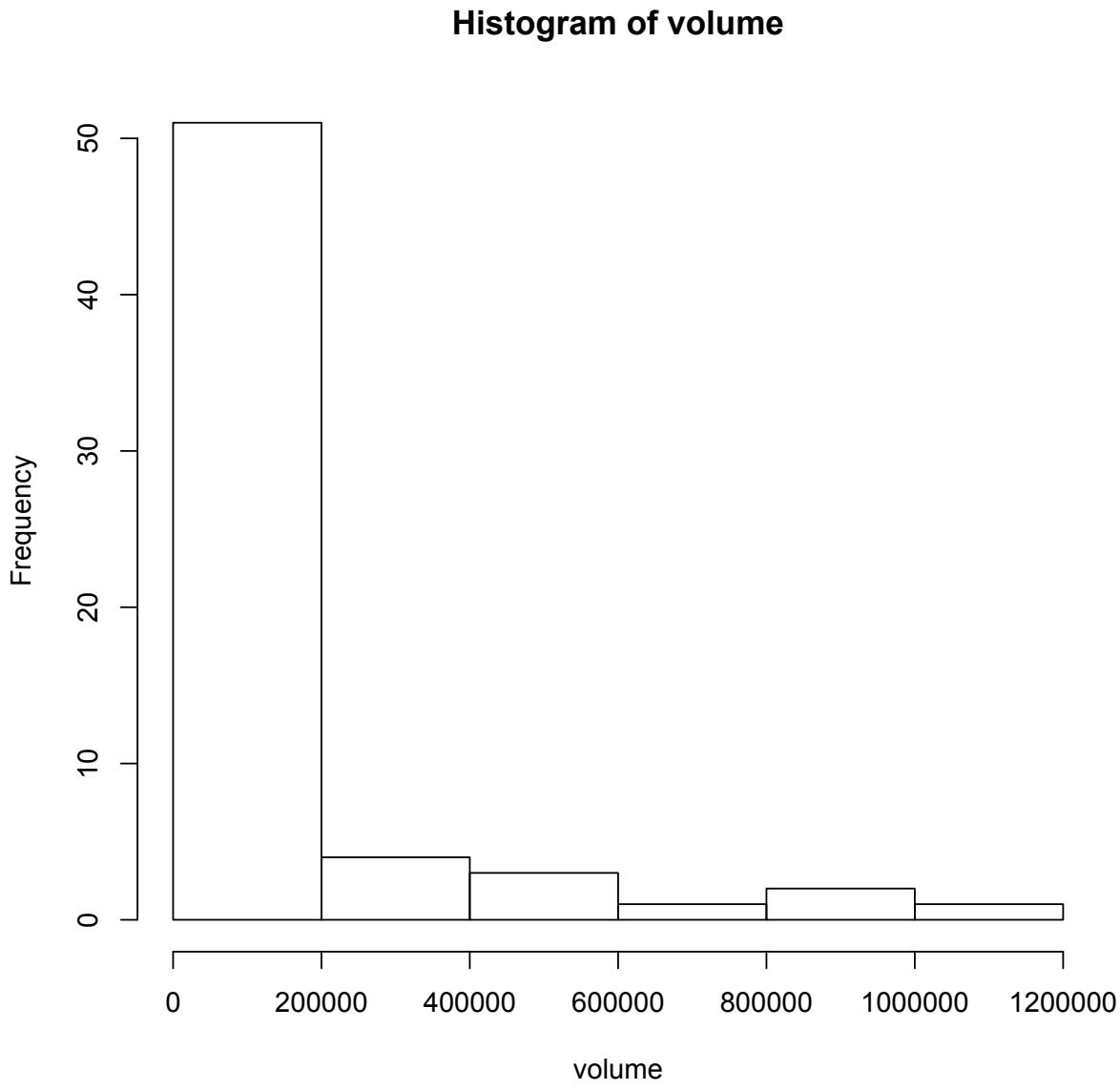


Figure C.4: Volume distribution across multiple systems

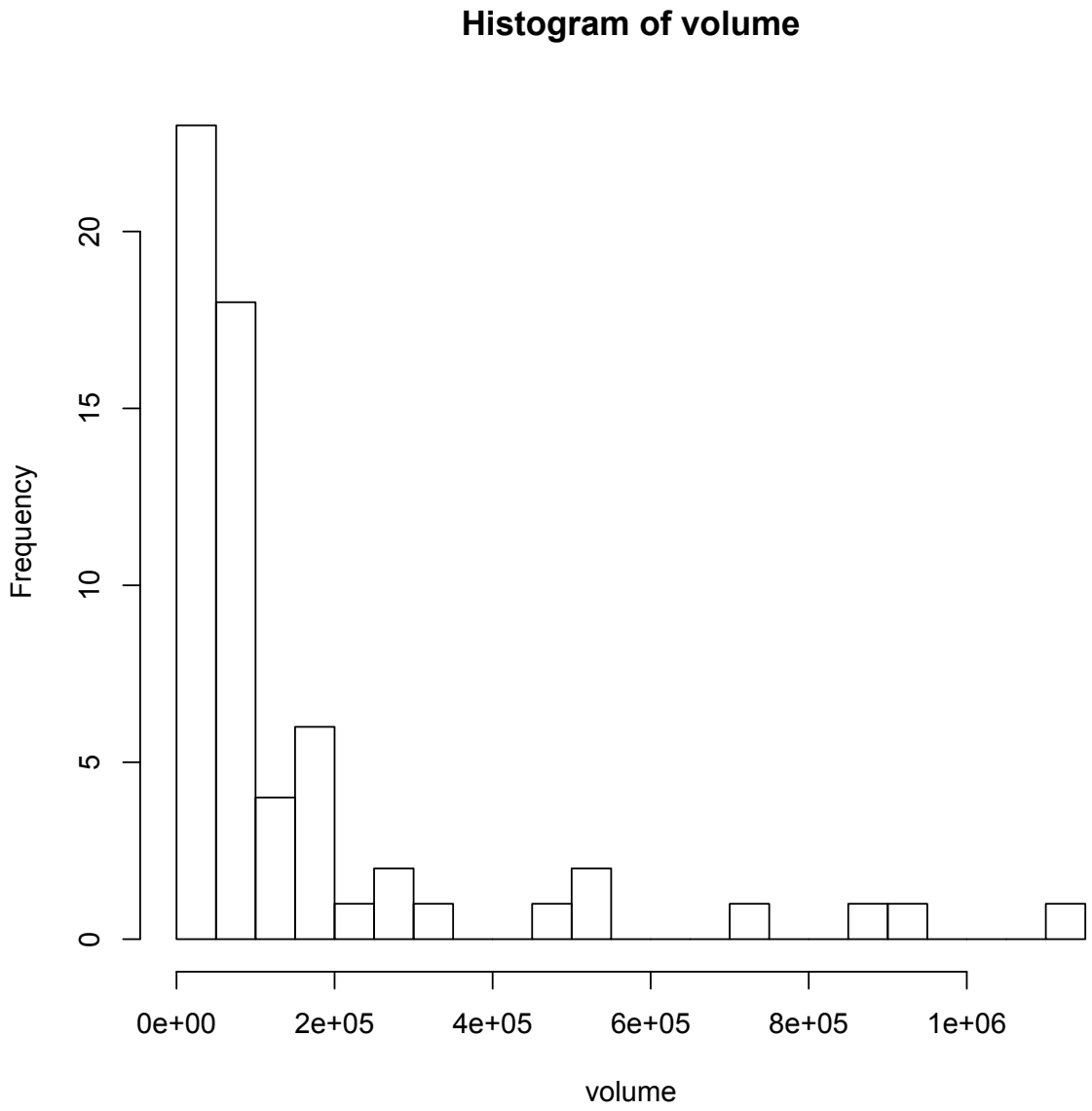


Figure C.5: Volume distribution across multiple systems (finer-grained)

C.2 Correlations

Volume Group	ρ	p-value	datapoints
0-100k	0.155345	0.3321	41
100k-500k	-0.2184244	0.4342	15
500k-1000k	0.6	0.35	5
1000k+	Insuficient datapoints		1

Table C.1: Stratified correlations between component count and complexity

Volume Group	ρ	p-value	datapoints
0-100k	0.1723327	0.2813	41
100k-500k	-0.01071429	0.9744	15
500k-1000k	0.6	0.35	5
1000k+	Insuficient datapoins!		1

Table C.2: Stratified correlations between dependency count and complexity

C.3 Plots

C.3.1 System A

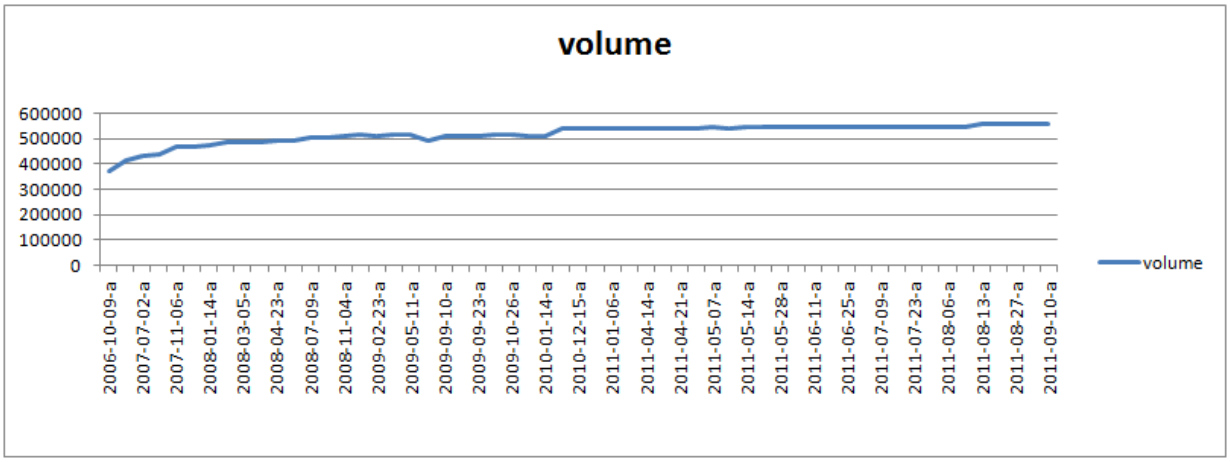


Figure C.6: Volume plot for system A

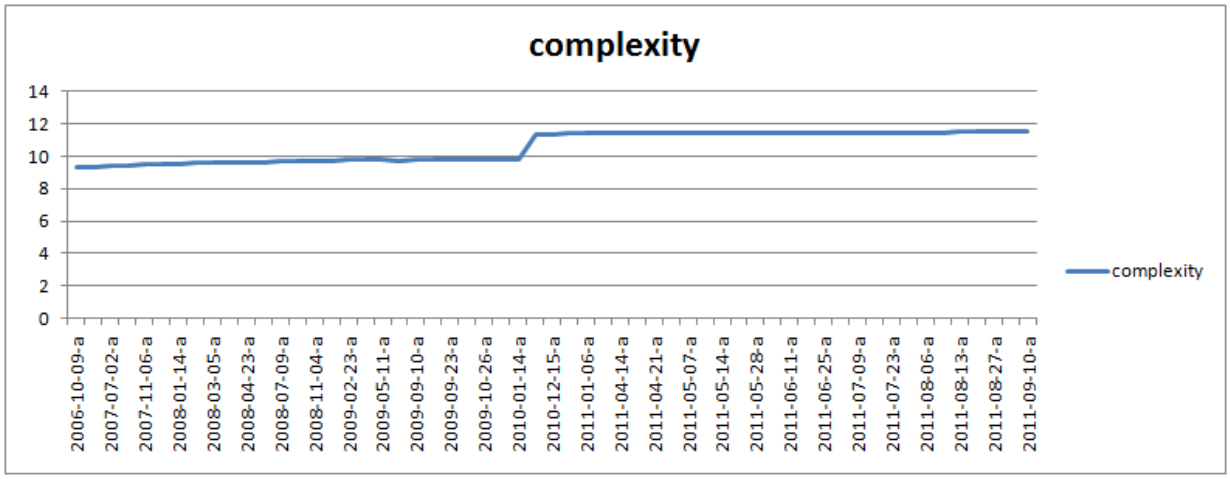


Figure C.7: Complexity plot for system A

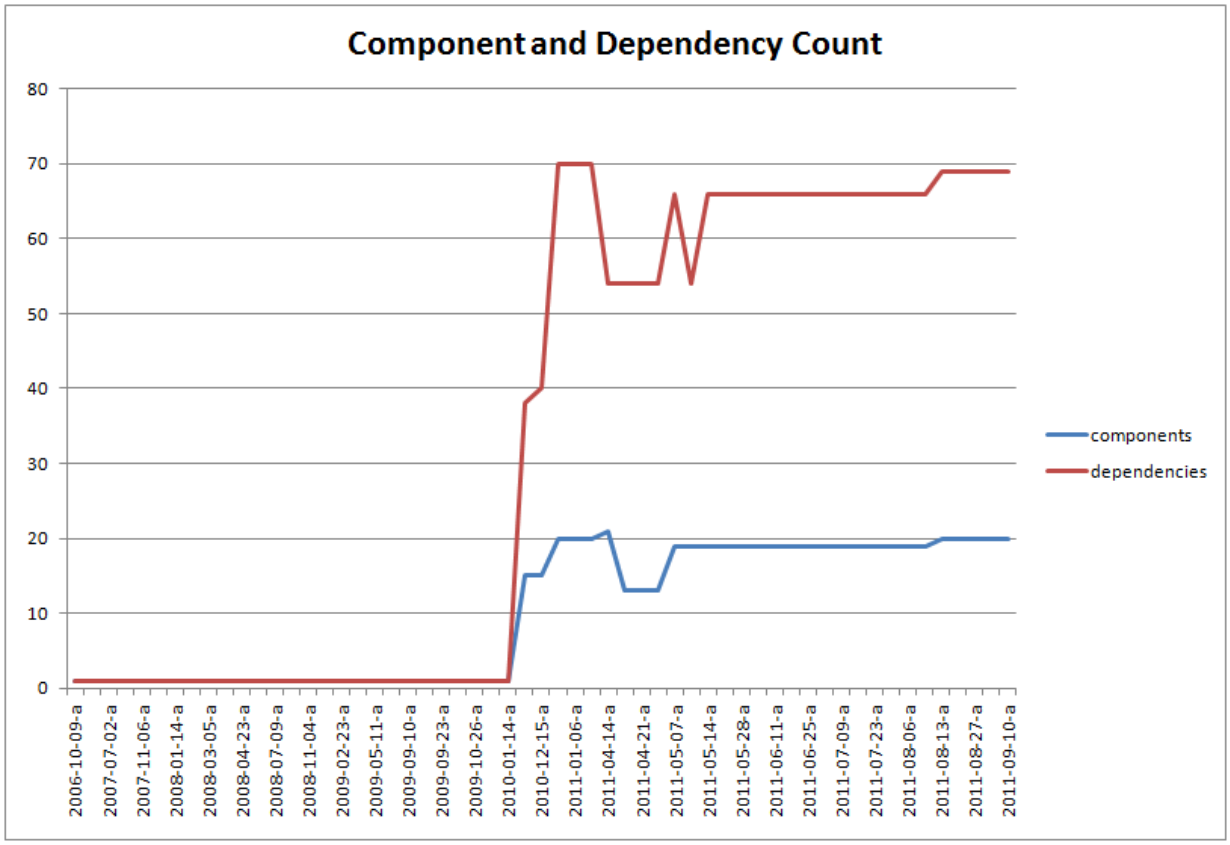


Figure C.8: Component and dependency count plots for system A

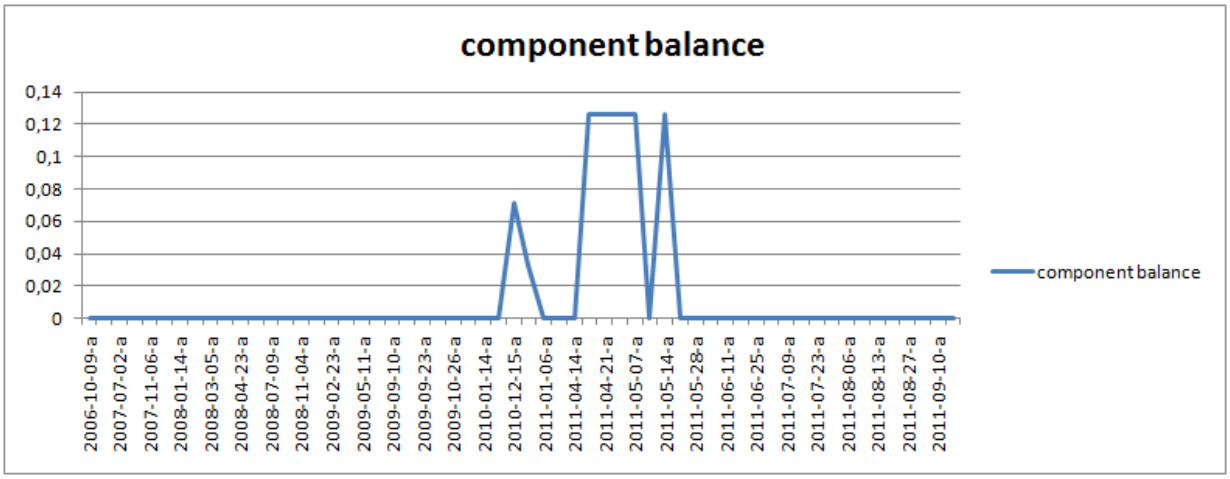


Figure C.9: Component balance plot for system A

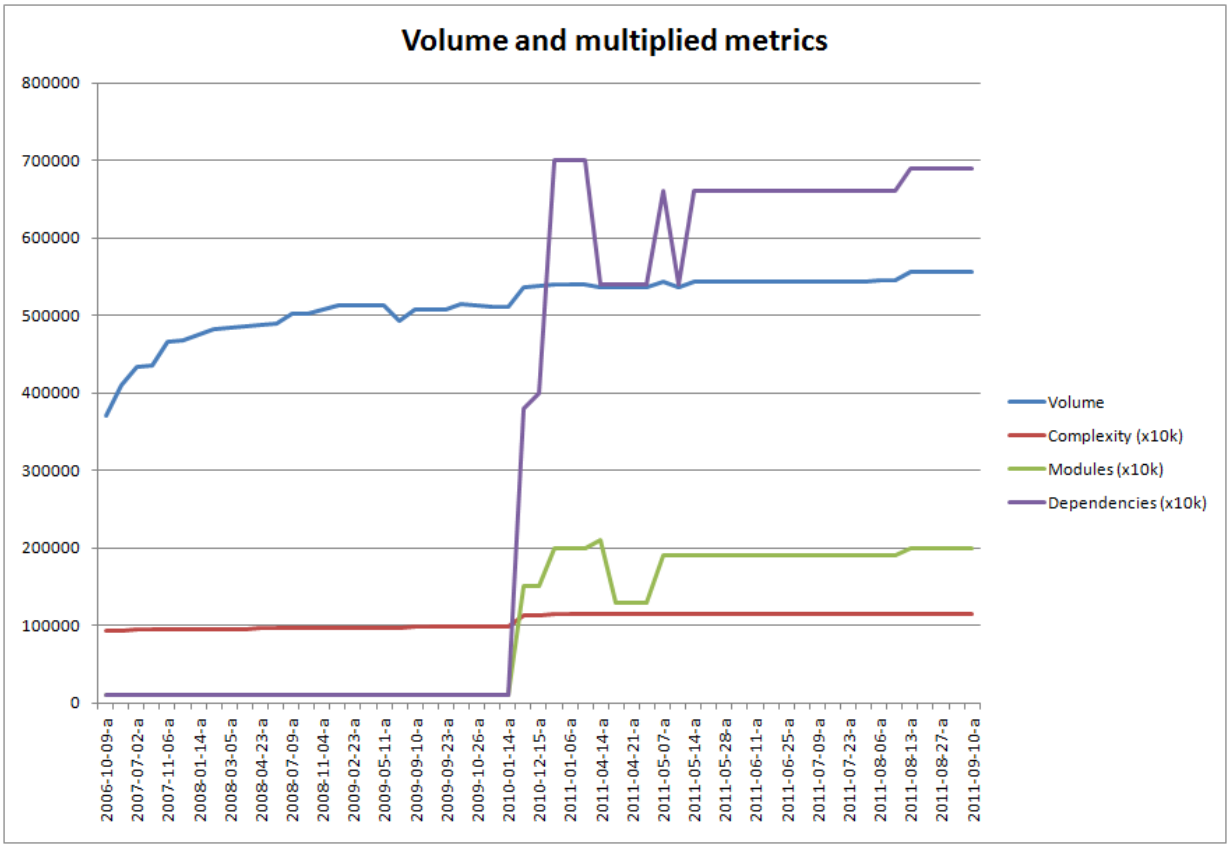


Figure C.10: Combined metric plot for system A

C.4 System B

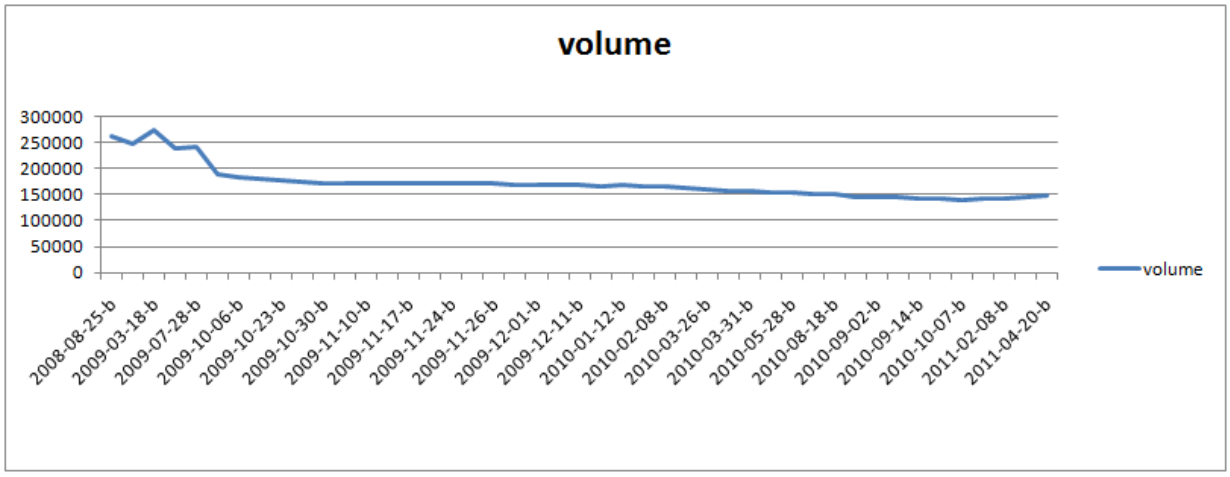


Figure C.11: Volume plot for system B

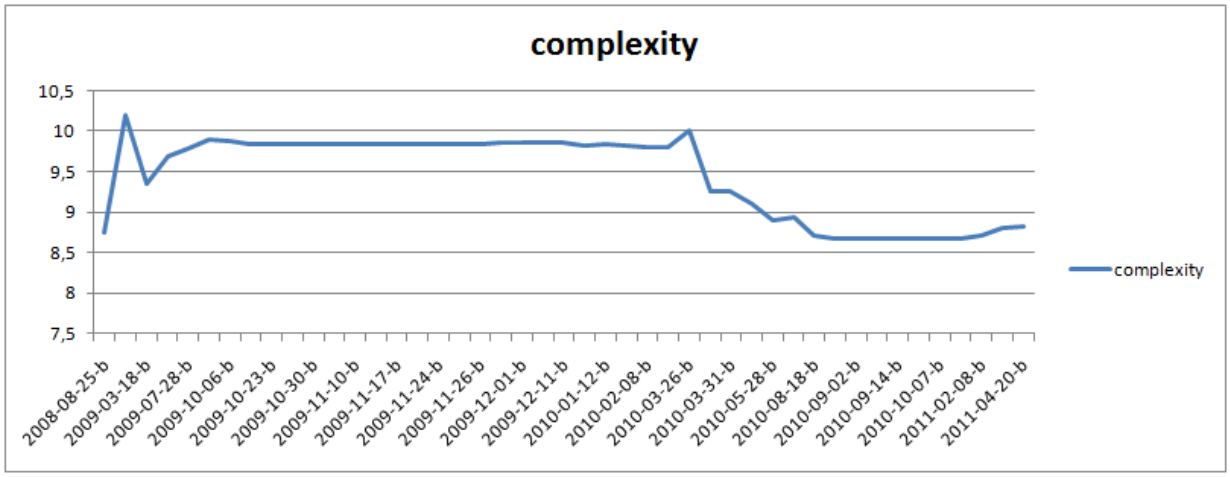


Figure C.12: Complexity plot for system B

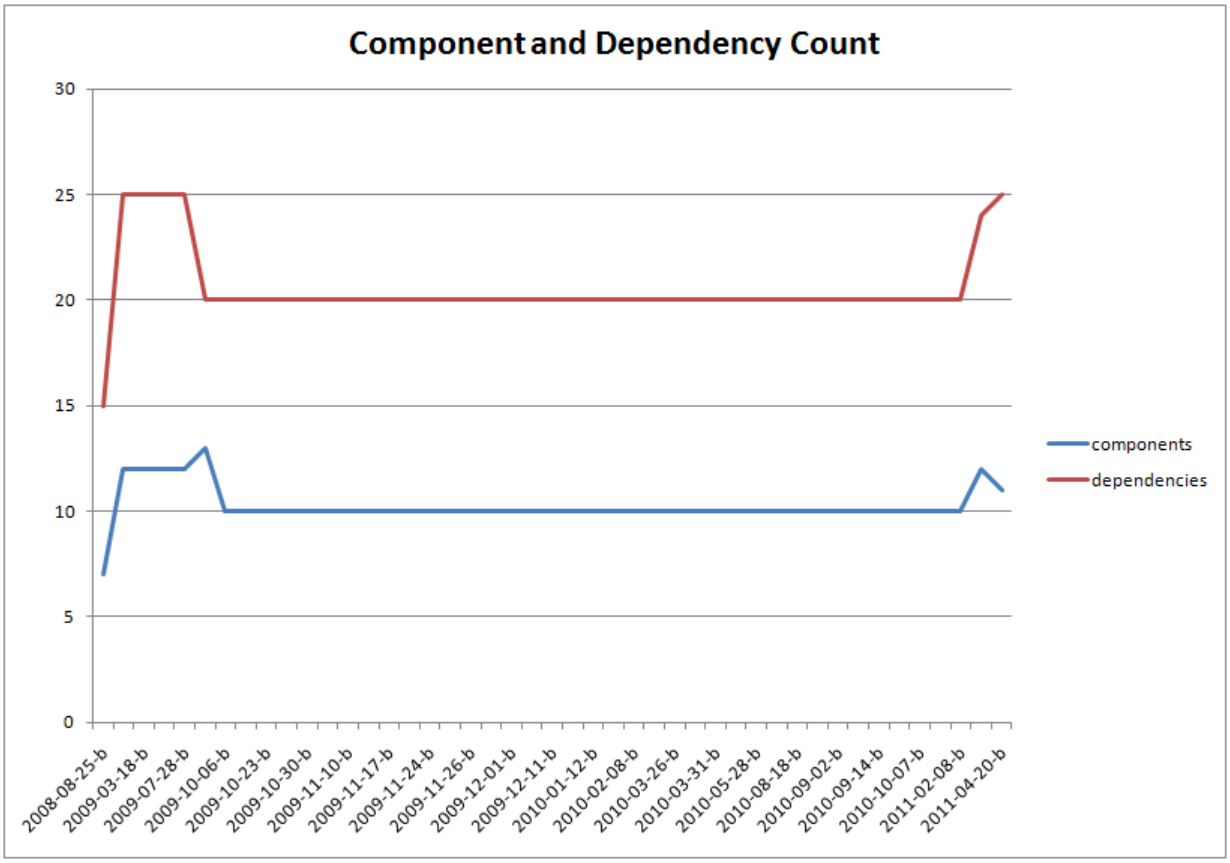


Figure C.13: Component and dependency count plots for system B

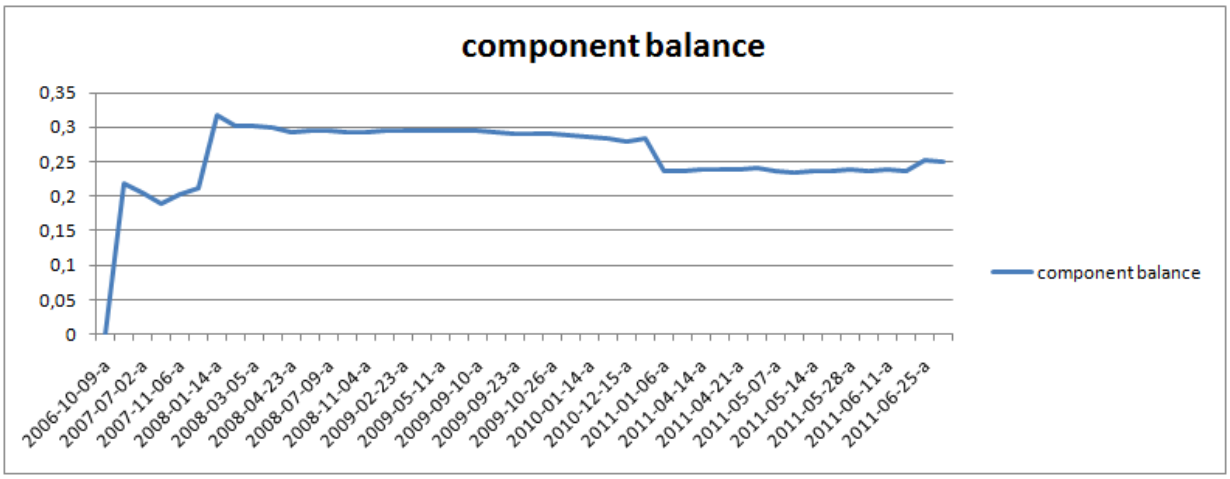


Figure C.14: Component balance plot for system B

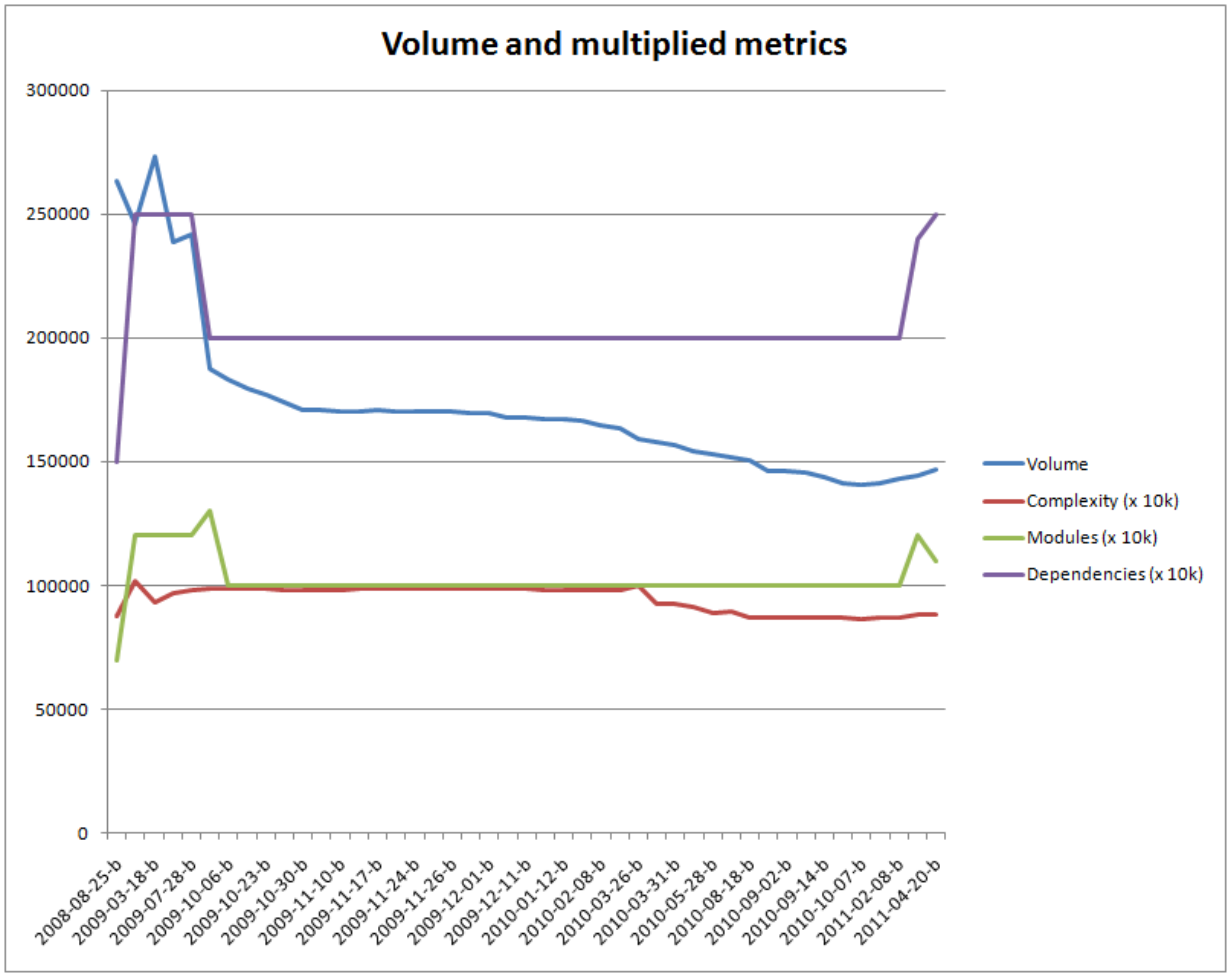


Figure C.15: Combined metric plot for system B