On the Interplay of Architecture and Collaboration on Software Evolution and Maintenance

A Thesis
Submitted to the Faculty of Drexel University
by
Sunny Wong
in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science
December 2010
Acknowledgements

Numerous people have been involved in the completion of this thesis. I am grateful for their support and acknowledge that, without them, I would not be where I am today. Words cannot express my gratitude to everyone who has helped me along the way—but here goes anyways. First and foremost, I would like to express my gratitude and appreciation to my advisor, Dr. Yuanfang Cai, without whom this thesis would not be possible. Dr. Cai’s guidance and support has been the driving force behind my success. I am indebted to her patient advising in leading me through this journey known as graduate school and academic research. The unique opportunity of working with her as been a truly unforgettable experience.

I would like to thank the other members of my thesis committee: Dr. Carliss Baldwin, Dr. Spiros Mancoridis, Dr. Ali Shokoufandeh, Dr. Giuseppe “Peppo” Valetto, and Dr. Tao Xie. The feedback and encouragement from my committee members have pushed me to look at my work from different perspectives and to work outside of my comfort zone. Their time and input has been greatly appreciated. I want to recognize Dr. Miryung Kim for her editorial advice and insights. She has helped me become a better writer in more clearly explaining technical details. I also would like to acknowledge my other collaborators: Dr. Alessandro Garcia, Dr. Jeff Gray, Dr. Claudio Sant’Anna, Dr. Kevin Sullivan, Yuanyuan Song, and Hyun Cho. It was a pleasure working with you all.

I want to thank my colleagues from Drexel University: Kanwarpreet Sethi, Archit Baweja, Georgi Simeonov, Kelly Blincoe, and Mike Dalton; whose efforts in getting various parts of this thesis published are greatly appreciated. My former colleague, Kanwarpreet “KP” Sethi, desires special recognition for his tremendous help. KP was always available to proofread, discuss research ideas, and help implement tools (especially MINOS and PARCAE)—even when he had his own thesis to work on. Thanks to the members of the SERG lab for the numerous “board meetings” and social events to keep me sane.

Finally, this research was supported in part by the Koerner Family Fellowship and by the National Science Foundation, under grants CCF-0916891 and DUE-0837665.
# Table of Contents

List of Figures ................................................................. iv  
List of Tables ........................................................................ vi  
Abstract ................................................................................ viii  
1. Introduction ........................................................................ 1  
2. Background ......................................................................... 5  
   2.1 Software Modularity ....................................................... 5  
      2.1.1 Canonical Modularity Theories ................................. 5  
      2.1.2 Modularity Models and Assessment .......................... 8  
   2.2 Evolutionary Coupling .................................................... 11  
      2.2.1 Association Rules Mining ....................................... 12  
      2.2.2 Other Evolutionary Coupling Derivation Techniques .... 13  
   2.3 Modularity Violations ................................................... 13  
      2.3.1 Conformance Checking .......................................... 14  
      2.3.2 Design Smells ....................................................... 16  
   2.4 Socio-Technical Congruence ......................................... 16  
      2.4.1 Conway’s Law ..................................................... 17  
      2.4.2 Coordination and Congruence ................................. 17  
      2.4.3 Empirical Studies ................................................. 19  
      2.4.4 Identifying and Addressing Coordination Mismatches ... 20  
3. Challenges .......................................................................... 22  
   3.1 Modularity ....................................................................... 22  
   3.2 Evolutionary Coupling ................................................... 22  
   3.3 Modularity Violations ................................................... 23  
4. Scalable Formal Model ....................................................... 24  
   4.1 Formalization of the Augmented Constraint Network .......... 27  
      4.1.1 Mathematical Formalization of ACN ....................... 27  
      4.1.2 Complexity of Dependency Analysis ...................... 29  
   4.2 Formalizing UML Class Diagram to ACN ....................... 30  
      4.2.1 Decisions in UML Classes and Interfaces .................. 31  
      4.2.2 Assumption Relations in UML ................................. 31
8.2 Model Conversion Tools .................................................................................. 136
  8.2.1 Design Model Converters ........................................................................ 137
  8.2.2 Reverse Engineering Tools ....................................................................... 138
8.3 Design Analysis Tools ..................................................................................... 139
  8.3.1 Parcae ........................................................................................................ 139
  8.3.2 Clio .............................................................................................................. 142
9. Conclusions ....................................................................................................... 144
Bibliography .......................................................................................................... 145
Vita ......................................................................................................................... 158
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example xADL Diagram</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Design Rule Theory Models</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Socio-Technical Congruence Example</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Graph Library Constraint Network</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Design Automaton</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Design Structure Matrix</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>UML Class to ACN Formalization</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>UML Dependency, Association, Aggregation, and Composition Relations</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>UML Dependency to ACN</td>
<td>33</td>
</tr>
<tr>
<td>4.7</td>
<td>Implication Graph</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>Compatibility Graph</td>
<td>37</td>
</tr>
<tr>
<td>4.9</td>
<td>Influence Graph</td>
<td>38</td>
</tr>
<tr>
<td>4.10</td>
<td>Modifying Vertex Cover</td>
<td>42</td>
</tr>
<tr>
<td>4.11</td>
<td>Modified Compatibility Graph Vertex Cover</td>
<td>44</td>
</tr>
<tr>
<td>4.12</td>
<td>Apache Ant UML Diagram</td>
<td>52</td>
</tr>
<tr>
<td>4.13</td>
<td>Maze Game Name Module DSMs</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Maze Game UML Class Diagram</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Maze Game DSM</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Partial Condensation Graph</td>
<td>67</td>
</tr>
<tr>
<td>5.4</td>
<td>Partial DR Hierarchy Graph</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>DR Hierarchy Algorithm Pseudocode</td>
<td>69</td>
</tr>
<tr>
<td>5.6</td>
<td>KWIC Design Structure Matrix</td>
<td>73</td>
</tr>
<tr>
<td>5.7</td>
<td>Dependencies Per Layer of Apache Ant DR Hierarchy</td>
<td>76</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparison between AL and SLDM Work Relationships</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Average $F_1$ per Release: Apache Ant</td>
<td>104</td>
</tr>
</tbody>
</table>
6.2 Average $F_1$ per Release: Hadoop Common ................................. 105
7.1 Class Diagram for Example Modularity Violation .......................... 116
7.2 Modularity Violation Identification Approach Overview .................. 119
7.3 Maze game design rule hierarchy .............................................. 121
7.4 Modularity Violation Detection Precision: Hadoop ......................... 126
7.5 Timing of Violation Detection (Hadoop) ....................................... 128
8.1 Tools Support Infrastructure: Model Conversion ............................ 134
8.2 Tools Support Infrastructure: Design Analyses ............................... 135
## List of Tables

4.1 UML Generalization to ACN ................................................................. 33  
4.2 UML Realization to ACN ................................................................. 34  
4.3 UML Owned Element Association to ACN ......................................... 35  
4.4 PWDR Running Time Comparison (in whole seconds) .......................... 48  
4.5 UML Formalization and PWDR Performance Results (in whole seconds) ................................................................. 50  
5.1 Proportion Tests ............................................................................... 79  
5.2 Comparison of Apache Ant and Eclipse JDT Project Measures ............. 81  
5.3 JDT Coordination Proportion Tests .................................................... 84  
5.4 JDT Coordination Proportion Tests (all tasks) ..................................... 85  
6.1 Example Lambda Sequences ............................................................. 97  
6.2 Stochastic Dependency Evaluation Subject System Information .......... 101  
6.3 Stochastic Dependency Evaluation Results ........................................ 103  
6.4 Wilcoxon Signed Rank Test Results \((k = 10)\) .................................... 106  
6.5 \(F_1\) Scores for \((\lambda, k)\) Pairs: Apache Ant ..................................... 107  
6.6 \(F_1\) Scores for \((\lambda, k)\) Pairs: Hadoop .......................................... 107  
6.7 \(F_1\) Scores for \((\lambda, k)\) Pairs: JEdit .............................................. 107  
6.8 \(F_1\) Scores for \((\lambda, k)\) Pairs: Log4J .............................................. 108  
6.9 \(F_1\) Scores for \((\lambda, k)\) Pairs: Eclipse .......................................... 108  
6.10 Hybrid Stochastic Dependencies Results: Apache Ant ....................... 109  
7.1 Characteristics of Modularity Violation Detection Subjects .................. 124  
7.2 Modularity Violations that Occurred at Least Twice in the Last Five Releases ................................................................. 125  
7.3 Characteristics of the Violations ...................................................... 129
For my parents and brother.
Abstract
On the Interplay of Architecture and Collaboration on Software Evolution and Maintenance

Sunny Wong
Advisor: Yuanfang Cai, Ph.D.

With the growing popularity of globally distributed teams in software development, software architectures must be designed and maintained with an organizational perspective in mind. Software architects often decompose systems into modules that are intended to be developed and maintained independently and in parallel. However, prevailing techniques for modularizing software designs often rely on intuition and experience, leading to modules that are not guaranteed to be independent tasks. Even when a design is well modularized at the beginning, the software can evolve in ways that deviate from the designed modular structure—making tasks that are initially designed to be independent, not actually independent. We lack the theories, models, and tools to address these issues, and support the parallel development and maintenance of software. Therefore, our research objectives are to develop design-centric theories and models to: (1) facilitate modularity assessment in terms of task parallelism, and (2) to identify design problems that harm task parallelism. To reach these objectives, in this thesis, we contribute a design-centric framework for maximizing concurrent work in software development and maintenance. Specifically, we contribute: (1) a scalable formal model of generalized software dependencies, (2) a model for automatically identifying tasks that can be assigned for parallel development, (3) an approach for identifying and prioritizing modularity violations that degrade task parallelism, and (4) a theory that takes the temporal dimension of revision history into consideration for better supporting software modularity analysis in the context of software evolution.
1. Introduction

With the growing popularity of globally distributed teams in software development, software architectures\(^1\) must be designed and maintained with an organizational perspective in mind. Software architects often decompose systems into modules that are intended to be developed and maintained independently and in parallel [96, 156]. However, prevailing techniques for modularizing software designs often rely on intuition and experience based on informal principles and theories (e.g. information hiding principle [156], design rule theory [22]), leading to modules that are not guaranteed to be independent tasks. Even when a design is well modularized in the beginning, the software can evolve in ways that deviate from the designed modular structure—making tasks that are initially designed to be independent, not actually independent. The problem is that prevailing design models and metrics (e.g. the unified modeling language [153, 164], coupling and cohesion [64, 179]):

1. are not designed for identifying independent modules for parallel work,
2. do not account for organizational structure, and
3. do not facilitate the early detection of design problems that harm task parallelism.

Therefore, our research objectives are to develop design-centric theories, models, and measures to: (1) facilitate modularity assessment in terms of task parallelism, and (2) to identify design problems that harm task parallelism.

Although the two prominent modularity theories (i.e., information hiding principle [156] and design rule theory [22]) emphasize the importance of designing software for independent task assignments, prevailing models and metrics are not intended for identifying parallel tasks and cannot quantitatively assess how well a design supports concurrent work. Studies on open source projects reveal that while some software systems are modularized to support large groups of distributed developers, many others are not. It has been recognized that a fundamental quality of successful software systems, both open-source and proprietary, is the ability to evolve through distributed and parallelized improvement of modules. Simply dividing software into components without considering organizational structure does not guarantee parallel development [156]. Software practitioners, however, often define a module as a function, class, or architectural component that does not guarantee independent development and maintenance. Additionally, empirical studies [53, 54, 69] reveal

---

\(^1\) Following Garlan and Perry [95], we define software architecture as “the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.”
a strong correlation between the alignment of design and organizational structures (called the socio-technical congruence), and both software quality and productivity. Yet we lack predictive models and measures to manipulate this socio-technical relation in order to maximize task parallelism.

Modularity violations (i.e., dependencies between modules that are intended to be independent) are often introduced during maintenance, creating technical debts [66, 125] in the system that need to be repaid at a later time. While modular software designs aim to localize maintenance tasks to a specific module, these technical debts expand the scope of modifying the software and increase maintenance costs. Two modules that are supposed to be independent may consistently change together, due to unexpected side effects caused by quick and dirty implementation. While approaches exist for identifying specific types of design problems (e.g., code clone [27, 73], design/code conformance [6, 148]), there are no approaches to identify modularity violations. Additionally, since the number of such violations may be cost prohibitive to entirely address, we need an approach that prioritizes certain violations such as to maximize the expected benefits from their resolutions. If a violation exists in a part of the system that is continually maintained and modified, it incurs more maintenance costs and is more important to resolve than a violation in a stable part of the system that is rarely modified. Prevailing approaches for identifying design problems often do not account for such a prioritization. However, addressing the most active violations first may be beneficial.

Revision history has been shown to be useful in many contexts of software analysis (e.g., software impact analysis [202, 204], coordination needs estimation [54, 142], software fault prediction [99, 150]). We have also found that some modularity violations are only revealed during software evolution. However, there is an important question that is often raised regarding the derivation of evolutionary dependencies [91] from revision history: how much historical data is needed for accurate analysis? We lack a formal theory for evolutionary dependencies, hindering the predicting abilities of these analysis techniques that rely on revision history. In particular, evolutionary dependencies do not account for the evolution of architecture and changes in modularity. That is, as a design evolves over time and the structure of dependencies change, events from the distant past of revision history may no longer reflect the current system’s state. There is a need for a formalization of evolutionary dependencies that accounts for this temporal dimension in order to improve the accuracy of design analysis techniques.

Towards attaining our research objectives, we make the following observations from the current state of art. The design rule theory states that once the design rules (i.e., stable decisions that decouple a design into modules) of a design are determined, their subordinate modules can be devel-
oped and maintained independently and in parallel. Based on this observation, we present a model that can enable the automatic identification of tasks for concurrent work. Next, we observe that if two components frequently change together to accommodate modification requests, but they belong to two separate modules that are supposed to evolve independently, there may exist a modularity violation between them. Therefore, we present an approach for identifying modularity violations based on unexpected modifications to independent modules. To address the problem that the use of evolutionary dependencies for design analysis does not account for design evolution, we present a formalization and generalization of evolutionary dependencies based on probability theory—in particular, a formalization that uses stochastic processes to incorporate a temporal dimension. Therefore, this thesis contributes:

1. A scalable, formal model of software dependencies to support the contributed approaches and a set of heuristics for converting a prevailing design model into our formal model.
2. A model for identifying independent tasks from a design, and assessing how well a design supports concurrent work.
3. An approach to identify modularity violations, and prioritize their severities.
4. A formal theory generalizing evolutionary dependencies to include a temporal dimension.

The problem of task parallelism in software development and maintenance is a large and difficult problem that crosses many disciplines (e.g. software design, human resource management). Because of this, the theories contributed in this thesis cannot address all aspects of the problem. We aim for these theories to complement theories in other disciplines, aiding software architects and managers in dealing with coordination in software development—especially distributed software development. For example, a manager may use our framework to identify tasks that can be assigned for concurrent development, while taking other factors (e.g. developer experience) into consideration.

Our contributed framework provides benefits for software architects/designers, developers, and managers. Architects can leverage our model of task independence to compare alternative designs in terms of how well they support concurrent development. The task independence model can also be used to assess how a system is evolving over time (e.g. are the number of parallelizable tasks increasing or decreasing relative to the size of the system). This evolutionary assessment can be useful for evaluating the quality of a software design prior to a company acquisition. Managers can use the independent tasks identified by our model in deciding how to assign tasks to teams for development. Both developers and managers can leverage tools that use our generalized evolutionary
dependency theory, for example, to perform change impact analysis. The estimated impact scope can provide managers with an estimate of effort needed to perform a task. To developers, the estimated impact scope can identify semantically related components (e.g. due to code clones) that may require their attention in fulfilling a change request. In order to meet deadlines, etc., developers may introduce technical debts into the software. Managers can take advantage of our modularity violation detector to locate these technical debts for resolution.

The remainder of this thesis is organized as follows: Chapter 2 provides background information on design rule theory, evolutionary dependencies, modularity violations, and socio-technical congruence. Chapter 3 describes the challenges faced by existing approaches, which this thesis directly addresses. To address scalability issues with our underlying design model, Chapter 4 presents approaches to support the adoption of our framework. Chapter 5 describes our model for identifying concurrent work from a software design. Chapter 6 describes our generalization of evolutionary dependencies. Chapter 7 describes our approach for identifying and prioritizing modularity violations. Chapter 8 presents the tools support for our framework and Chapter 9 concludes.
2. Background

In this chapter we review the most relevant and significant areas of prior work that are related to our contributions. We first describe two prevailing modularity theories (i.e., information hiding and design rule theory) and their supporting design models and metrics. Next, we describe the current techniques for the derivation of evolutionary coupling. Then, we discuss modularity violations and technical debt. Finally, we discuss socio-technical congruence and its impact on productivity and software maintainability.

2.1 Software Modularity

Software modularity, or the structure of dependencies within and between software components, has been extensively studied due to its widespread influence over the construction, maintenance, and quality of software. In fact, Baldwin and Clark [21] argue that modularity increased the rate of innovation in the computer industry and is the key factor for its success. Creating highly modular software, which minimizes dependencies between components, has been shown to have many benefits. According to Stevens et al. [179]:

The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other modules. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous “ripple” effects where changes in one part cause errors in another, necessitating changes elsewhere, giving rise to new errors, etc.

Below we first describe two prevailing modularity theories. Then we survey models and metrics for assessing design modularity.

2.1.1 Canonical Modularity Theories

In this subsection, we describe the highly influential information hiding principle and the more recent design rule theory.
Information Hiding Principle

One of the most influential concepts in modularity is Parnas’s information hiding (IH) principle [156]. Parnas argued that modular decomposition should be assessed based on potential changes to the system. Given a change to the requirements or operating environment, we should examine its impact scope in terms of how many modules would need to change. In order to minimize this impact scope, Parnas advocated that modules only expose stable interfaces and hide parts that are likely to change. By only exposing to other modules the details that are not likely to change, modules can be modified more easily without causing significant impact to other parts of the software. Not surprisingly, information hiding forms the foundation of many modern approaches for reducing component dependencies, such as data encapsulation and interfaces in object-oriented languages.

Another benefit of information hiding is that once the interfaces have been defined, modules can be developed independently and in parallel. Prior to information hiding, software modules were often identified based on flowcharts of control logic. Parnas explicitly defines a module as an independent task assignment and not a subprogram, in order to emphasize the idea of parallel work. Ghezzi et al. [96] corroborate this idea in their textbook:

If a design is composed of highly independent modules, it supports the requirements of large programs: independent modules form the basis of work assignments to individual team members. The more independent the modules are, the more independent the team members can proceed in their work.

Based on information hiding, Hoffman [114] offers guidance on the design of module interfaces, stating that interfaces should be consistent, essential, general, minimal, and opaque. Consistency prescribes that interfaces follow standard naming conventions, parameter passing styles, exception handling strategies, etc. that allow predictability. Essentiality implies that needless and duplicate features are omitted. Generality means that the interface’s applicability is not limited to its initial purpose, including room for future expansion. Minimality separates independent features and splits an interface that has services that may be used separately by clients. Opaqueness provides information hiding by encapsulating details that may change.

For completeness sake, we briefly discuss a slightly different view on IH modularization. The recent idea of open implementation [127] states that modules should not hide all implementation details from other modules, as in the case of IH. Instead, Kiczales et al. argue that when designing the interface to a module, we should allow clients some control over its implementation strategy.
This idea leads to a range of openness strategies:

1. no selection of implementation strategy
2. clients provide usage pattern information for automatic implementation strategy selection
3. clients select from a predefined set of implementation strategies
4. client provides their own implementation strategies that the module adopts

These open implementation strategies are already in use in popular object-oriented languages. For example, the third and fourth strategy are used in the java.util.List interface of the Java standard library, with available implementation strategies such as ArrayList and LinkedList. This theory does not contradict information hiding, but rather generalizes the concept of an interface to include implementation strategy selection. Further generalizing the concept of an interface, we next introduce the idea of a design rule.

**Design Rule Theory**

Baldwin and Clark [22] generalized information hiding in their design rule theory by proposing the concept of design rules (DRs) as generalized interfaces. Design rules are defined as stable design decisions that decouple otherwise coupled decisions; these are the decisions that are made before and respected by subordinate design decisions. We emphasize that Baldwin and Clark’s concept of a design rule is different from the concept of a rule used in other areas (e.g. “do no create clones,” “avoid cyclic dependencies”); rather, they are essentially generalized interfaces between components. Hence, design rules include not only programmatic interfaces but also non-software-structure-related decisions such as requirement specifications, operating environment conditions, standardized naming conventions, etc. As a concrete example, in the design of a graph library, the decision on whether clients will use dense or sparse graphs dictates whether to use an adjacency matrix or adjacency list representation and algorithm. This density requirement acts as a design rule dominating the data structure and algorithm decisions; in addition, the subordinate decisions do not influence the design rule, since clients will not change the density of their graphs solely because of a developer’s affection for a particular data structure. Similarly to information hiding, design rules are expected to be stable and once they have been defined, modules can be further designed and developed independently.

The idea behind design rule theory is that modularity in design creates economic value in the form of real options. Design rules enable the splitting of modules into independent components, thus creating economic value in the form of options. Each module provides the option (but not
the obligation) of being substituted with a better version without disturbing other parts of the system. Substituting modules with better implementations, analogously follows the information hiding principle idea that by hiding details that are likely to change, we can change a module’s implementation without causing significant impact on the rest of the system. Additionally, the ease of substituting modules allows for rapid evolution of software in order to fulfill customer demands; the more options a design has, the easier and faster it can evolve. Sullivan et al. [184] were the first to apply option theory to software design analysis. We further discuss the use of real options for assessing modularity in the next subsection.

2.1.2 Modularity Models and Assessment

Due to the importance of modularity, many approaches have been proposed to model and assess the modularity of a system. In this subsection, we describe some prevailing models and metrics for software design modularity.

Design Models

The unified modeling language (UML) [153, 164] serves as the prevailing de facto modeling language for software designers in both academia and industry. This box-and-line style modeling language includes many different diagram types, with the component diagram and class diagram commonly used for representing architecture and design structure, respectively. Both these diagram types use boxes to represent components or classes, and lines to represent various relations between components and classes, with different line styles and arrowheads denoting different types of relations. Since UML is a rather informal modeling language, researchers have proposed techniques of formalizing it (e.g. Evans et al. [83], Baresi and Pezzè [23]). In addition, due to the various UML diagram types presenting different views of a software design, several researchers have explored approaches to ensure consistency between diagrams (e.g. Straeten [181], Egyed [75, 76]).

Similar to UML, most architectural description languages (ADLs) use boxes to represent components and lines to represent relations between components. Unlike UML, ADLs model high-level software structure (architecture) and ignore implementation details. Medvidovic and Taylor [141] survey a number of ADLs and present a framework for comparison. Example ADLs include C2 [140], Wright [8], and xADL [68]. Figure 2.1 shows an example xADL diagram modeling two peer-to-peer clients accessing data through a central server. Unlike some ADLs, xADL uses boxes to depict both components and connectors.
Baldwin and Clark [22] adopted the design structure matrix (DSM) as the design representation in their theory. DSMs were created by Steward [180], developed by Eppinger [82], and has been applied to various engineering disciplines. Figure 2.2(a) shows an example DSM. A DSM is a square matrix in which rows and columns are labeled with dimensions where design decisions are made (commonly called variables); a marked cell models that the decision on the row depends on the column. Modules are represented as blocks along the diagonal; for example, row 1 forms a module, and rows 2 and 3 forms another module. Design rules are captured in a DSM by asymmetric dependencies that decouple modules. For example, all the variables depend on the design rule (column 1) but it does not depend on anything else.

To address the problem that manually constructing DSMs is error prone and subject to ambiguity, Cai and Sullivan [49] proposed the augmented constraint network (ACN) as a formal counterpart to the DSM, formalizing the concept of design rules and enabling automatic DSM derivation. An ACN consists of a set of variables (representing dimensions where design decisions with finite domains are made), a set of constraints upon the variables (restricting the space of legal designs), and a set of dominance relation pairs (formalizing design rules). Figure 2.2(b) shows an example ACN that models the design of a graph library (overly simplified for sake of illustration). The ACN shows that three design decisions need to be made in designing this library: the density of graphs clients will use, the data structure to represent the graphs, and the type of algorithm to use. The first constraint says that the decision to use an adjacency matrix data structure assumes that the client needs dense graphs. The dominance relation pairs indicate that the decisions for what data structure and algorithm to use, cannot influence the client’s requirement on density of graphs (the design rule). In other words, clients will not change the density of graphs they use just because the library is designed with a certain data structure or algorithm. In providing a formal foundation for
marking the cells of a DSM, Cai and Sullivan [49, 50] proposed the concept of *pairwise dependency relation* (PWDR). Formally, a variable $y$ is pairwise dependent on a variable $x$ if and only if “$y$ must be changed in some minimal restoration of consistency to the constraint network, which was broken by a change in $x$.”

![Design Structure Matrix]

(a) Design Structure Matrix

![Augmented Constraint Network]

(b) Augmented Constraint Network

**Figure 2.2: Design Rule Theory Models**

**Metrics and Measures**

The most well known measures of modularity are coupling and cohesion [64, 179]. Coupling measures the density of dependencies between components; the more and stronger the dependencies exist between two components, the more coupled they are. Since higher coupling between two components tends to require more effort in maintenance of the components and more coordination between developers, lower coupling is a desirable trait in designs. On the other hand, cohesion measures the density of dependencies within a component; the more and stronger the dependencies within a component, the more cohesive it is. Semantically, cohesion measures the functional relatedness of elements within a component. Since higher cohesion within a component tends to ease the identification of related elements for performing maintenance, higher cohesion is desirable in designs. Using coupling and cohesion heuristics, several tools have even been developed to automatically cluster source code entities into modules (e.g. Hutchens and Basili [116], Sarkar et al. [167], Arch [169], Bunch [137, 144, 145]).
The informal nature of coupling and cohesion has led many researchers to define quantitative measures and metrics to better assess and compare designs. Chidamber and Kemerer’s [57] suite of six metrics for object-oriented (OO) software is among the most referenced metrics based on coupling and cohesion. These metrics quantify source code modularity by measuring properties of object-orientation such as inheritance tree height, interface size, and number of methods called between two classes. Briand et al. [35,36] present a survey and theoretically validation of several coupling and cohesion OO metrics at the source code level.

Sullivan et al. [184] were the first to apply Baldwin and Clark’s [22] real options theory to software design analysis. Modeling Parnas’s canonical keyword in context (KWIC) [156] system using a DSM, they showed that by clustering environment parameters as the left-most block in the DSM, they could visually capture Parnas’s information hiding criteria. Additionally, they applied Baldwin and Clark’s net option value (NOV) analysis on both KWIC designs that Parnas described and reported consistent conclusions as Parnas on the superior modularity of the information hiding design.

NOV analysis has been performed by numerous researchers since then. Lopes and Bajracharya [133], and Sullivan et al. [183] applied NOV in the context of aspect-oriented (AO) programming. Bahsoon and Emmerich [20] used it to assess architectural stability. Cai et al. [48] used NOV, as well as several other measures that they introduced, to compare the OO and AO versions of designs patterns. They reported consistent results with the source code modularity analysis of Hannemann and Kiczales [103]. Calculating NOV, however, relies upon several qualitative parameters that are not derivable from software structure, such as substitution cost and replacement benefit. The recently proposed, independence level (IL) [172,173] measurement quantifies the portion of a design that can be substituted without disturbing other parts, allowing it to be calculated more accurately than NOV. Sethi et al. [173] applied the IL measure to the component diagrams of several releases of a software product line and reported consistent results with an earlier source code analysis [86].

2.2 Evolutionary Coupling

The two major categories of software dependencies are structural (also called syntactic) dependencies and evolutionary (also called logical or multi-version) dependencies. Structural dependencies are analogous to traditional program dependence graphs [115] that represent control (e.g. function A calls function B) and data (e.g. data structure D is modified by function A and used by function B) dependencies. Evolutionary dependencies were introduced by Gall et al. [91] and has been developed
in the software evolution literature. Instead of looking at programming constructs, evolutionary dependencies are based on files that are changed together in development tasks, as identified by revision control system commits. By leveraging revision history, evolutionary dependencies are able to detect semantic coupling between components that are absent from structural dependencies. Because of the identification of semantic coupling, evolutionary dependencies may be more effective than structural dependencies for certain modularity analyses (e.g. software impact analysis [202, 204], coordination needs estimation [54, 142], software fault prediction [99, 150]).

There are various data mining approaches for deriving evolutionary dependencies from revision history data. The most common techniques are based on association rules mining [2, 202, 204]. Recently, other data mining techniques (e.g. sequential pattern mining [4]) have been applied to derive evolutionary dependencies. In the following subsections, we describe association rules mining and briefly present some other techniques for deriving evolutionary computing. A more thorough survey of data mining techniques, in the context of software evolution, can be found in the work of Kagdi et al. [120].

2.2.1 Association Rules Mining

Ying et al. [201, 202] and Zimmermann et al. [203–205] were among the first to apply association rules mining to the derivation of evolutionary dependencies. An association rule is of the form $a \Rightarrow b$, where $a$ and $b$ are each non-empty sets of software elements (called itemsets). We refer to the union of $a$ and $b$ as a co-change pattern. An evolutionary dependency is said to exist if the heuristic values for an association rule are higher than certain thresholds.

The two main heuristics used in deriving evolutionary dependencies are support [202] and confidence [204]. Support is defined as the number of commits/transactions that $a$ and $b$ occur together in revision history. Confidence is defined as support, divided by the number of times that $a$ occurs (with or without $b$). Intuitively, confidence can be seen as the maximum likelihood estimate of the conditional probability $\Pr(b \mid a)$ [113, 196]. While other heuristics (e.g. $\chi^2$ correlation [40], conviction [41], bond [155]) have been proposed for association rules mining, they are rarely used in deriving evolutionary/logical dependencies.

---

1. We use the term software element to generically refer to constructs at different levels of granularity (e.g. file, class, function).
2. In the data mining community, support [2] is defined as the percentage of transactions that contains $a$ and $b$. When used for evolutionary dependencies, support is defined as the absolute number of transactions [202, 204].
As an example, we consider the association rules $A \Rightarrow B$ and $B \Rightarrow A$ for files $A$ and $B$. File $A$ appears in 100 transactions; file $B$ occurs in 20 transactions; and they appear together in 10 transactions. There is a support of 10 for the rule $A \Rightarrow B$ and a support of 10 for the rule $B \Rightarrow A$. There is a confidence of 0.5 for rule $B \Rightarrow A$ (i.e., file $A$ is logically dependent on file $B$) and a confidence of 0.1 for rule $A \Rightarrow B$ (i.e., file $B$ is logically dependency on file $A$).

The two popular algorithms to derive association rules for evolutionary coupling are Apriori [3] and Frequent-Pattern (FP) growth [101, 102]. The FP-growth algorithm can identify frequently occurring itemsets (called frequent patterns) to construct association rules. Using frequent patterns eliminates the need to examine all combinations of software elements in constructing associations and thereby reduces the number of association rules processed. Both Apriori and FP-growth are offline algorithms (i.e., they do not support incremental changes with the addition of new transaction). While online algorithms (e.g. Hidber [111]) have been proposed for association rules mining, their use in deriving evolutionary coupling is limited.

### 2.2.2 Other Evolutionary Coupling Derivation Techniques

Rather than finding itemsets that frequently occur in transactions, sequential pattern mining [4] deals with finding frequently occurring sequences of itemsets. For example, if file $B$ always appears in a transaction immediately after a transaction with file $A$, sequential pattern mining would suggest that file $B$ is logically dependent on file $A$ (i.e., a change to file $A$ requires a compensating change to file $B$). Similarly to association rules mining, heuristics (e.g. support, confidence) are also used in sequential pattern mining to derive sequences. Sequential pattern mining has been used to identify evolutionary coupling by various researchers (e.g. Gall et al. [91]).

A sequence identified by sequential pattern mining can intuitively be seen as a cause and effect relation from changing software elements. Various other techniques have also been considered for identifying such cause/effect relations from revision history data. For example, Ceccarelli et al. [55] use the bivariate Granger causality test [98] and Bouktif et al. [31] use dynamic time warping [131].

### 2.3 Modularity Violations

Modularity violations are technical debts [66, 125] that impact not only the development and maintenance of the software, but also the end users. Recent studies [53, 69] show a correlation between software defects and modularity violations that are not recognized by developers. As interest from
the technical debt on the design structure, these extraneous dependencies enlarge the impact scope of making changes to software. On the organizational structure, the larger impact scope requires more developers to be involved in coordinating the maintenance process. Hence, the identification and removal of modularity violations is critical for enabling parallel work among developers.

While researchers have defined numerous types of software dependencies, we find the definition by Spanoudakis and Zisman [176] to be general enough to include all these types. Hence, unless specified otherwise, we refer to a dependency as a directional relationship between two software entities \((e_1, e_2)\) such that the \(e_1\) relies on the existence of \(e_2\) to perform its operations or changes to \(e_2\) must be reflected on \(e_1\). Using this definition we see that some design and implementation issues such as code clone can be seen as modularity violations. For example, cloned code exhibit a semantic dependency between each other. If a problem is found in one of the clones, the other clones also need to be fixed.

The relation between software dependency structure and defects has been widely studied. Many empirical evaluations (e.g. Selby and Basili [171], Cataldo et al. [53]) have found that modules with lower coupling are less likely to contain defects than those with higher coupling. Various metrics have been proposed (e.g. Chidamber and Kemerer [57]) to measure coupling and failure proneness of components. The relation between evolutionary coupling and defects has also been recently studied. Cataldo et al.’s [53] study revealed a strong correlation between density of evolutionary coupling and failure proneness. Fluri et al.’s [87] study shows that a large number of evolutionary coupling relationships are not entailed by structural dependencies. While the purpose of these studies are to statistically account for the relationship between software defects and dependencies, they do not locate modularity violations, which may cause design decay and software defects. These existing studies often define measures that suggest a component may contain functional defects. However, modularity violations may incur maintenance costs without affecting the functionality of the software and may not be revealed by these existing measures.

### 2.3.1 Conformance Checking

Conformance checking approaches (e.g. Reflexion model [148], ArchJava [6]) are traditionally used for detecting extraneous dependencies. Most conformance checking approaches use the concepts of a prescriptive architecture [187], which is the design as intended by the designers, and a descriptive architecture [187], which is the design as realized in the code. Differences between the prescriptive and descriptive architectures, known as architectural drift [187], occur due to the introduction of
unexpected dependencies or missing dependencies between components. Architectural drift not only makes it difficult to understand and maintain code, but can also degrade software quality [69].

The Reflexion model [148, 149] is one of the most well-known code conformance checking approach. Designers describe their prescriptive architecture using a box-and-line style notation and define mappings from components in this putative model to source code entities. Based on these mappings, the Reflexion tool extracts syntactic dependencies (e.g., call graph) from the source code and reports any architectural drift. Both Krikhaar et al. [129] and Tran et al. [189, 190] also proposed architectural repair techniques very similar to the Reflexion model. In addition to identifying conformance violations, they also describe techniques for resolving issues, by changing either the prescriptive or descriptive architecture. Sangal et al. [166] describe a similar conformance checking approach, using DSMs reverse engineered from source code using Lattix.\(^3\) Using Lattix, the user can specify which classes should not depend on (i.e., syntactically refer to) which other classes. The tool alerts the designer if a predefined constraint is violated.

Since manually defining mappings from the prescriptive architecture to source code entities can be difficult and tedious, many extensions to the Reflexion model have been proposed. One of the most well-known of these extensions is the work of Christl et al. [59]. They presented an incremental, semi-automated approach based on the Reflexion model. Their user-interactive approach identifies potential clustering of source code entities based on coupling and cohesion heuristics. Users still need to associate the source clusters with prescriptive components, but mapping time is reduced since there potentially are fewer clusters than code entities.

Some conformance checking approaches allow more detailed granularity than checking presence/absence of dependencies. Pattern-Lint of Sefika et al. [170] allows the specification of design patterns to ensure conformance. Eichberg et al. [79] takes this idea further and uses a declarative language, DataLog, to codify more general design intents. By codifying the prescriptive architecture in DataLog, they can check for specific allowable connectors between components. For example, if software elements \(A\) and \(B\) are only allowed to communicate via message passing but a developer accidentally makes a direct method call, this dependency violation would not be identified by most approaches that use box-and-line representations of the prescriptive architecture.

\(^3\)http://www.lattix.com
2.3.2 Design Smells

Fowler [88] describes the concept of “bad smell” as a heuristic for identifying designs to be refactored. Example bad smells include code clone and feature envy (when a class excessively uses methods of another class). Garcia et al. [93] proposed several architecture-level bad smells. Bad smells are often due to modularity violations and various approaches have been proposed to identify bad smells. To automate the identification of bad smells, Moha et al. [146] presented a tool, Decor, and a domain specific language (DSL) to automate the construction of design defect detection algorithms. For example, to detect spaghetti code, the authors describe (using the DSL) that potential defect locations contain long methods, methods with no parameters, no inheritance, use of global variables, and no polymorphism. Based on such a description, their framework generates an algorithm that can analyze Java code looking for such properties.

Several other techniques (e.g. Tsantalis et al. [191–193]) automatically identify bad smells that indicate a needs of refactoring. For example, Tsantalis and Chatzigeorgiou’s technique [192] identifies extract method refactoring opportunities using static slicing. Higo et al. [112] proposed the Aries tool to identify possible refactoring candidates based on the number of assigned variables, the number of referred variables, and dispersion in the class hierarchy. A refactoring can be suggested if the metrics for the code satisfy certain predefined values.

Detection of some specific bad smells such as code duplication has also been extensively researched. For example, Baxter et al. [27] and Tairas and Gray [185] use abstract-syntax trees to detect clones. Kamiya et al. [124] detect clones using a token-based comparison technique. Recent approaches (e.g. Gabel et al. [90], Krinke [130]) try to identify semantically related code clones by applying similarity measures to software dependency graphs.

2.4 Socio-Technical Congruence

Factors such as cost concerns, company mergers, and acquisitions have made multi-site software development commonplace [104]. It is common for an open source project to have developers from multiple continents. However, globally-distributed software development suffers from a myriad of communications and coordination issues. In this section, we survey studies on socio-technical congruence and some of the problems caused by communications and coordination in distributed software development.
2.4.1 Conway’s Law

The importance of organizational communication patterns in software development has long been recognized. Conway [65] suggested (in what became known as Conway’s Law or the mirroring hypothesis [63, 134]) that the structure of a system mirrors the structure of the organization that designed it. Using Conway’s Law, several researchers (e.g. Bowman and Holt [32, 33]) have reverse engineered the architecture of software based on developer information. Bowman and Holt found this ownership architecture to be quite useful, including its identification of experts for components and discovery of non-functional dependencies.

MacCormack et al. [134] found empirical evidence in support of Conway’s Law in their comparison of open source and commercial projects. For each commercial project they examined, they compared it against a similar open source project (e.g. Linux against XNU). They reported that since developers in the commercial projects were collocated, they were able to build more tightly coupled software components. On the other hand, the open source projects that had less coupled components also had developers that are highly distributed. Further support for Conway’s Law in their work, comes from investigation into one of the open source projects, Gnumeric, that had much higher coupling than any of the other open source projects. Upon further examination, they found that two developers had been contributing more than half of the work to Gnumeric. Hence, since Gnumeric was not as distributed as the other open source projects, its component coupling was significantly higher.

Around the same time as Conway, Brooks [42] noticed the difficulties of coordination in software engineering practice. In what became known as Brooks’s Law, he described how adding more people to an already late project further slows it down, due to project coordination and communications overhead. In fact, Curtis et al. [67] found, through an empirical study, that communication and coordination is one of the most difficult and prevalent problems in large-scale software engineering.

2.4.2 Coordination and Congruence

Although the significance of communications and organization structure has been long recognized, the term socio-technical congruence [54] is fairly new. Socio-technical congruence measures the alignment of task dependencies to coordination activities. Figure 2.3 illustrates an example congruence problem, with Figure 2.3(a) showing coordination activities between developers and Figure 2.3(b) showing task (e.g. source code artifact) dependencies, grouped by developer assignment.
We see from the figure that task 5, which is assigned to developer D, depends on task 2, which is assigned to developer B, but D does not coordinate with B. On the other hand, developer D communicates with developer C even though there are no dependencies between their tasks. No matter what reason causes this mismatch, it may affect both the quality of the software being developed and the time to finish development. For example, developer D may speak a different language than B and uses developer C as an intermediary to get the necessary information. In that case, every time D needs some information from B, s/he first contacts C and waits for C to get this information from B. Not only does this degrade D’s productivity, but it also degrades C’s productivity due to time being a proxy between B and D. Additionally, since D is getting information interpreted through C, s/he may misunderstand something that B meant and inadvertently introduce a problem into the code.

In comparing design and organizational structures, Cataldo et al. [54] proposed the concept of coordination requirements (CRs). A coordination requirement matrix is a developer-by-developer matrix that represents the extent to which pairs of developers need to coordinate their work. They defined the matrix as \( C_R = T_A \cdot T_D \cdot T_A^T \), where \( T_D \) is a task-by-task matrix of task (design structure) dependencies, \( T_A \) is a developer-by-task matrix of task assignments, and \( T_A^T \) is the transpose of \( T_A \). They compared the coordination requirement matrix against actual communications that occurred to calculate the socio-technical congruence.
In forming a theory of coordination, Herbsleb et al. [109,110] postulate that “the irreducible
inter-dependence among software development tasks can be thought of as a distributed constraint
satisfaction problem (DCSP) where coordination is a solution to the DCSP” [53]. They argue that
the density and structure of dependencies between developers affect coordination success, quality of
the software system, and the productivity of the development organization.

2.4.3 Empirical Studies

Herbsleb et al. [106,108] report that it takes about two and a half times longer for distributed
developers to complete a task than for collocated developers, due to cross-site communications.
Unlike Brooks’s Law, excessive communications overhead is not the only communications problem
in multi-site development; lack of developer communications also presents an issue. In fact, Allen [9]
reports that, in a study of engineering organizations, when engineers’ offices are placed 30 meters or
more apart, their communications drop to levels of people with offices many miles apart.

Various researchers have performed empirical studies on the relationship between software ar-
chitecture and communication structure. For example, Sosa et al. [175] found that dependencies
between software components correlated with frequency of developer communications. Avritzer et
al. [16,17] used design structure matrices to model the modularity of their software architecture, and
reported that the architecture and actual coordination structures were highly consistent. Amrit and
van Hillegersberg [11] studied the design structure and developers’ mailing list of the JBoss open
source project. They found that when the design dependencies dramatically increased (measured
using the propagation cost and clustered cost of MacCormack et al. [135]), the amount of coordina-
tion of developers also significantly increased. On the other hand, Bass et al. [25] studied several
organizations that were unable to adequately handle the coordination needs incurred by certain task
dependencies. They argued that some architectural decisions may create tasks dependencies that
organizations may not be able to handle in terms of coordination needs; however, they also state
that these problematic architectural decisions are specific to organizational situations, and therefore
they did not generalize the type of decisions to avoid.

Recent studies (e.g. Cataldo et al. [52,54]) show that when socio-technical congruence is high,
development tasks are accomplished more quickly. For example, Cataldo et al. [52] studied 39 months
of development of a large distributed system, in an organization with 114 developers distributed over
three locations. During that period, the time it took developers to resolve each modification request

4We use the terms change request and modification request interchangeably.
was recorded. The researchers calculated coordination requirements based on their earlier work [54] using evolutionary coupling for design structure dependencies. They report that when the actual communication structure matched the coordination requirements, modifications tasks were resolved more quickly.

The relation between design structure and organizational structure has also been studied in other engineering fields. For example, Gulati and Eppinger [100] report that in several case studies there was a strong interplay between product architecture and organizational structure. Organizations exhibited mechanisms of changing one structure that affected the other, leading to a co-evolution of the two structures.

2.4.4 Identifying and Addressing Coordination Mismatches

Herbsleb and Grinter [105] argue that “the more cleanly separated the modules, the more likely the organization can successfully develop them at different sites.” Hence, a prevailing technique for addressing (or more accurately, preventing) coordination mismatch is to design modular architectures, under the assumption that reducing component dependencies also reduces the need for module developers to communicate. Various empirical studies (e.g. Morelli et al. [147], Sosa et al. [175], de Souza et al. [71]) support this assumption. In software engineering, Parnas [156] was one of the first to suggest creating modular designs to support independent work. Baldwin and Clark [22] argue that modular design structure leads to modular task structure, where each module can be assigned to a team and work can be performed independently of each other. According to their design rule theory, the decoupling effects of design rules allows this independent work.

The Adriadne [69, 70] tool visually shows developers who they may need to coordinate with, based on translating design dependencies to coordination requirements. This tool mines software repositories for dependency information, constructing call-graphs and identifying maintainers of the dependent modules. Developers are then able to visualize, in a graph structure, their coordination requirements and the relative strengths of those requirements. Adriadne also allows developers to “zoom” to different abstraction levels of the source (e.g. class, method, package). Amrit and van Hillegersberg [10] developed a similar tool to Adriadne, called TESNA, that works on the design level rather than the source code level. Sarma et al. [168] also developed a similar tool, called Tesseract.

Valetto et al. [195] developed the socio-technical software network (STSN) for analyzing socio-technical congruence. This network structure consists of two graphs: $G_P$ representing the communications structure and $G_S$ representing source code artifacts and their dependencies. The graphs are
on two different planes, and edges across planes (from $G_P$ to $G_S$) represent task assignments. They presented an algorithm for detecting socio-technical congruence mismatches from a STSN, based on mirroring of edges between the communications and artifact dependencies graphs. Using the STSN, Ehrlich et al. [78] found that STC gaps (i.e., lack of communications between pairs of developers with task dependencies) occurred more often with distributed developers than collocated developers, and that increased number of gaps were associated with more code changes.
3. Challenges

In this chapter, we discuss some of the challenges faced by existing approaches for enabling parallel development of large scale software systems. The contributions of this thesis directly address these challenges.

3.1 Modularity

Despite their benefits, the information hiding principle [156] and design rule theory [22] remain informal, with only heuristics to guide practitioners in modularizing software. Prevailing modeling languages are not designed for identifying parallel tasks, nor do they account for organizational structure. Instead, software practitioners often define a module as a function, class, or architectural component that does not guarantee independent work. We need a model that can assist architects and managers in identifying, from a design, independent tasks that can be assigned for parallel work.

3.2 Evolutionary Coupling

Recent studies [143,199] show, when limited version history is available, evolutionary dependencies can be inaccurate in performing impact analysis. On the other hand, when extensive revision history is available, the architecture may have evolved and dependencies that previously existed between components may no longer exist. Since the support heuristic never decreases, once support passes the minimum threshold, the dependency between the components continues to exist. To address this issue, the confidence heuristic can be used in conjunction with support. However, as the number of transactions becomes large, confidence only minimally changes with each transaction. Or more precisely (where $\sigma$ is the support and $M$ is the number of transactions):

$$\lim_{M \to \infty} \left( \frac{\sigma}{M + 1} - \frac{\sigma}{M} \right) = 0$$

and

$$\lim_{M \to \infty} \left( \frac{\sigma + 1}{M + 1} - \frac{\sigma}{M} \right) = 0$$

Besides for using confidence, some researchers (e.g. Minto and Murphy [142]) simply limit the length of historical data (e.g. only transactions from the latest year) used for analysis. However, selecting the length of data to use is informal and can lead to the same problems as when only limited version history is available.
Additionally, timestamps of transactions are generally ignored when deriving association rules (i.e., transactions are considered as an unordered set rather than an ordered sequence). Whether a co-change pattern occurs in a recent transaction or in one from the distance past, it contributes the same to support and confidence for deriving evolutionary coupling. Sequential pattern mining [4] and similar techniques maintain the order of transactions but only to identify causal relationships, not to account for design evolution. Without a formal theory, we cannot easily incorporate this temporal aspect of transactions into the derivation of evolutionary dependencies, and obtain more accurate design analyses.

### 3.3 Modularity Violations

While conformance checking techniques (e.g., Reflexion model [148]) are often used for detecting extraneous dependencies in software, modularity violations often consist of semantic dependencies that cannot be identified by these existing approaches. For example, cloned code may be semantically dependent on each other; if a bug is found in one clone, the other instances may need to be changed also. However, prevailing conformance checking techniques do not address such semantic dependencies. Additionally, certain design smells may be difficult to properly identify using only syntactic information. Al-Ekram et al. [5], for example, describe how “accidental” clones often appear in source code due to certain component interaction protocols, which may cause prevailing approaches to report many false positives.

We lack a theory to uniformly identify various types of modularity violations. Additionally, since the number of extraneous dependencies may be cost prohibitive to entirely address, we need a way to prioritize them based on the interest they accrue. In other words, the modularity violations that are frequently impacting maintenance tasks should be addressed first, since they are likely to continue affecting maintenance efforts.
To support the adoption and use of our theories and techniques in software engineering practice, we contribute (1) a scalable formal model of software design, and (2) a set of heuristics for formalizing the popular unified modeling language (UML) class diagram model into our formal model.

Our formal model is based on the *augmented constraint network* (ACN) model of Cai and Sullivan [46, 49]. As described in Section 2.1, the ACN models design decisions as first-class constructs and expresses how decisions make assumptions upon each other using logical constraints. By basing our formal model on the ACN, we can leverage various existing design-level automated modularity and changeability analyses, (e.g. design volatility analysis [172,173]). ACN modeling has been used to formalize Parnas’s information hiding principle [156,184] and the notion of design rules in Baldwin and Clark’s modularity theory [22,46,49]. It has been used to automate and quantify Parnas’s changeability analysis [49,156]. ACNs have also been used to generate design-level DSMs for numerous projects [48–50,173,183]. Our prior work shows that by comparing the ACN-derived DSM at design level and the DSM generated from the source code of the same project, we are able to identify implicit dependencies and improper implementations that caused modularity deviation [117].

Despite the potential utility of ACN modeling shown in recent work, there are two major obstacles of leveraging ACN analysis in practice. First, constructing design-level ACNs are not straightforward, requiring the designer to model software design in the form of decisions and model their relations using logical expressions. Second, the automated analysis techniques enabled by ACN models rely upon constraint solving, which is difficult to scale, similar to other formal models used in model checking [62] and formal specification [118,178]. Different from other formal models (e.g. Alloy [118]), that are designed to check software properties such as consistency, ACN modeling focuses on expressing assumption relations among decisions, and aims to reason about dependencies and modularity properties. Addressing the scalability issue in such a model is particularly important because comprehension difficulty and modularity decay become prominent and relevant only when the software is of a certain scale. The current approach of deriving the pairwise dependency structure from an ACN (i.e., the automatic derivation of design structure matrices) requires not only finding one satisfying solution to the constraint network but enumerating all the solutions, making the approach even harder to scale. We present two contributions to directly address these problems.
First, we formalize the widely used unified modeling language (UML) [153, 164] class diagrams as decisions and their constraints, and thus enable the automatic derivation of ACN. UML is a well-understood and widely-used modeling technique, using boxes and lines to model classes and their relations. Important design decisions and their relations are embodied by these models. For example, a class can be viewed as having at least two design dimensions, a public one that is visible to other decisions and a hidden one that hides the secret of the class. The relations among classes, such as inheritance and association, determine whether and how a change in one class will influence other classes. Our purpose is to reap the best from both sides, enabling software designers, who are familiar with UML modeling, to take advantage of modularity analyses enabled by ACN and DSM modeling. On the other hand, the size of a UML class diagram can easily scale to the extent that comprehension of the overall system becomes difficult [77]. Our approach allows the designers to view the structure of a large-scale UML model in a modular way.

Second, to make the ACN model and the associated analysis techniques applicable to large-scale software systems, we identify some common characteristics possessed by the majority of existing ACN models, representing medium to large scale real software systems, as well as the ACNs derived from UML class diagrams. Based on these observations, we contribute an approach to generate dependency relations from a class of more restricted, but highly representative, ACNs, reducing the complexity for these restricted formal models from NP-complete to polynomial.

Our approach is based on the following observations. When investigating the dependency, and hence modular, structure of a software system, especially for large-scale software systems, we mainly care about whether one dimension makes assumptions about other dimensions, but not how. As a result, a variable in a software ACN often has only two values: \textit{orig} to model a current decision, and \textit{other} to model an unknown possible decision that is different from the current one. This simplification makes ACNs different from other formal models that detail the states of each component. Second, the major relation we care about is the \textit{assumption} relation that can be expressed using logical implications.

We call an ACN that exhibits these two characteristics as a \textit{binary augmented constraint network} (BACN, pronounced \textit{bacon}). In this chapter, we present an algorithm to derive pairwise dependency relations for BACNs without constraint solving. These two characteristics allow us to automatically derive the \textit{assumption} dependency relation from UML class diagrams of large-scale software systems.
It is worth noting that the number of dependencies derived from an ACN transformed from a UML class diagram is much larger than the number of dependencies discovered from corresponding source code using prevailing reverse-engineering tools. For example, we identified 622 pairs of dependencies from a UML-transformed ACN for our Parcae while using Lattix, only 271 dependencies were identified. The majority of the differences are due to the fact that ACNs make implicit and indirect dependencies explicit.

In this chapter, we focus on addressing the scalability issue for large ACN models. To solve a constraint network with hundreds or thousands of variables is impractical. Nevertheless, software ACN models with this size are often BACN. We studied 70 ACNs, acquired from published work or from on-going projects in modeling real software systems, which model multiple versions of heterogeneous software systems. Some of these ACNs were manually constructed, while others were automatically derived from other design models. Of these 70 ACNs, 58 of them shared the two BACN characteristics described above. The two remaining ACNs were of very small scale, with less than 20 variables.

Our evaluation aims to assess whether our approach can generate pairwise dependencies correctly, and whether it can be applied to real, large-scale software systems. We compute the dependency relations, hence the design structure matrices, from all ACNs available in literature, and also from ACNs automatically derived from the UML class diagram reverse engineered from large-scale software systems. We demonstrate the correctness of our approaches and show that the time needed often reduces from hours to seconds. The results show that this approach has the potential to make constraint-based design modeling and automated dependency analysis techniques applicable to real software systems.

The rest of this chapter is organized as follows: Section 4.1 presents a mathematical formalization of the augment constraint network model, as background to this work. Section 4.2 describes the heuristics for formalizing a UML class diagram into an ACN model. Section 4.3 formalizes the BACN model and Section 4.4 introduces the polynomial-time dependency derivation algorithm. Section 4.5 describes a change impact analysis technique based on BACN modeling. Section 4.6 details our evaluation for both the UML formalization and dependency analysis algorithm. Section 4.7 discusses threats and future work. Section 4.8 briefly reviews some related work and Section 4.9 concludes.

1 http://www.lattix.com
4.1 Formalization of the Augmented Constraint Network

In this section, we describe the theoretical background of our work. In particular, we formalize the augmented constraint network (ACN) and pairwise dependence relation (PWDR) [46, 49]. The small example ACN used in this section is also used in Section 4.4 to illustrate our algorithm. Additionally, we contribute a proof of the computational complexity for deciding PWDR.

4.1.1 Mathematical Formalization of ACN

An ACN consists of three parts: a finite-domain constraint network, a dominance relation, and a cluster set (not discussed in this chapter). A constraint network is a tuple $N = (V, U, d, C)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of variables representing design dimensions where decisions are needed. Then $d : V \rightarrow 2^U$ is a mapping of variables to a finite set of valid domain values. Hence, $U$ is the universe of domain values for all variables. Lastly, $C$ is a finite set of constraints upon the variables. Figure 4.1 shows an example constraint network that models the design of a graph library (simplified for illustration). For example, the first constraint says that the decision to use an adjacency matrix data structure assumes that the client needs dense graphs. We use the term binding to refer to specifying the value of a variable in a constraint (e.g. $ds = matrix$, $density \neq dense$).

Variables:
- $density : \{dense, sparse\}$
- $ds : \{matrix, list\}$
- $algo : \{matrix, list\}$

Constraints:
- $ds = matrix \Rightarrow density = dense$
- $ds = list \Rightarrow density = sparse$
- $algo = matrix \Rightarrow ds = matrix$
- $algo = list \Rightarrow ds = list$

Figure 4.1: Graph Library Constraint Network

The dominance relation $D \subseteq V \times V$ models an asymmetric dependency relation among decisions, formalizing Baldwin and Clark’s concept of design rules [22]. We define two dominance relation pairs for our graph example: $(ds, density)$ and $(algo, density)$, indicating that the decisions for what data structure and algorithm to use, cannot influence the client’s requirement on the density of graphs to be used. In other words, clients will not change the density of the graphs they use solely because of a developer’s affection for a particular data structure or algorithm.
We define a solution to a constraint network as a mapping \( s : V \rightarrow U \) such that all variables are mapped to valid domain values \( \forall v \in V \ s(v) \in d(v) \) and all constraints are satisfied. Each solution to a constraint network is a valid design for the software modeled. For any constraint network, we define \( S \) as the set of all its solutions. Given two solutions \( s, s' \in S \), we use the notation \( s - s' \) to represent the set of variables that are assigned different values by the two solutions—or formally, \( s - s' = \{ v \in V \mid s(v) \neq s'(v) \} \). We use the notation \( s \setminus s' \) to represent the set of variables and values in \( s' \) that are different from \( s \)—or formally, \( s \setminus s' = \{ (v, u) \in V \times U \mid s'(v) = u \land s'(v) \neq s(v) \} \). It is important to note that while \( s - s' \) is commutative, \( s \setminus s' \) is not (i.e., \( s \setminus s' \neq s' \setminus s \)).

From the constraint network and dominance relation we can derive a non-deterministic finite automaton, called a design automaton (DA) [49]. The set of solutions \( S \) to the constraint network form the states of the DA, and each transition models a change in a design decision. Given an initial design \( s \in S \), the transition function \( \delta(s, v, u) \) denotes the valid designs resulting from changing variable \( v \) to domain value \( u \) in \( s \). Since changing \( v \) to \( u \) may violate some constraints, the value of other variables may need to change to maintain a valid design. However, if a variable \( v' \) must change to restore solution satisfiability but \( v' \) dominates \( v \) (i.e., \( (v, v') \in R \) meaning that changes to \( v \) cannot force \( v' \) to change), then such a change is considered invalid. In addition, transitions only show the destination states that differ minimally from the initial state. Formally, we define \( \delta(s, v, u) \equiv \{ s' \in S \mid s'(v) = u \land \neg \exists \tilde{s} \in S \tilde{s}(v) = u \land s \setminus \tilde{s} \subset s' \setminus \tilde{s} \land (\forall v' \in (s - s') (v', v) \notin D) \} \).

Maintaining satisfiability through minimal perturbation form the concept of pairwise dependence relation (PWDR). Cai and Sullivan [50] define PWDR as a set \( P \subseteq V \times V \), such that if \( (u, v) \in P \) (i.e., \( v \) depends on \( u \)) then \( v \) must be changed in some minimal restoration of consistency to the constraint network, which was broken by a change in \( u \). To formally define PWDR, we first define a mapping \( \Delta : S \times V \rightarrow 2^S \) to be the set of states directly reachable from an initial state \( s \) by changing a variable \( v \) to any valid domain value—or formally, \( \Delta(s, v) = \bigcup_{u \in d(v) \setminus \{s(v)\}} \delta(s, v, u) \). From this, we formally define the PWDR set \( P \) such that \( (v, v') \in P \) if and only if \( \exists s, s' \in S s' \in \Delta(s, v) \land v' \in (s - s') \).

![Figure 4.2: Design Automaton](image-url)
Figure 4.2 shows the DA for the small graph library example. Since there are only two satisfying solutions to the constraint network, the DA contains two states. For simplicity, we refer to the states as dense for \{density = dense, ds = matrix, algo = matrix\}, and sparse for \{density = sparse, ds = list, algo = list\}. There is only one valid transition from dense to sparse (i.e., by changing density to sparse) because changing either ds or algo would require changing density, which the dominance relation forbids. Since changing density to sparse requires changing both ds and algo to maintain satisfiability, both ds and algo are pairwise dependent on density. The marks in the first column of the design structure matrix (DSM) in Figure 4.3 show these dependencies. In a design structure matrix, the columns and rows are labeled with design variables. A cell is marked if there is a dependency between the decision on the row to the decision on the column. The remaining marks between algo and ds are due to the fact that every time one of them changes, the other does also.

![Design Structure Matrix](image)

**Figure 4.3: Design Structure Matrix**

### 4.1.2 Complexity of Dependency Analysis

The problem of deciding a pairwise dependency relation from a constraint network is NP-complete in general. We prove this by a Karp reduction from the constraint satisfaction problem (CSP) \[136\]. Given a finite set of variables \(V'\), a finite domain of values \(U'\), and a finite set of constraints \(C'\), the CSP problem is to decide whether there exists an assignment of domain values to variables that satisfy all the constraints. We reduce an instance of CSP to a PWDR decision problem: given an ACN and two variables \(a\) and \(b\), decide whether there is a pairwise dependence \((a, b)\).

From a CSP instance, we construct an ACN instance by adding two additional variables \(V \equiv V' \cup \{\alpha, \beta\}\), two additional domain values \(U \equiv U' \cup \{true, false\}\), and an additional constraint \(\alpha = true \Leftrightarrow \beta = true\). Since all variables from the CSP instance can be any domain value, we let \(\forall v \in V' d(v) = U'\). We restrict the domains of \(\alpha\) and \(\beta\) to be only \{true, false\}. Hence, the added constraint guarantees that \(\alpha\) and \(\beta\) will be assigned the same value in all solutions, and changing the value of \(\alpha\) in any solution requires changing \(\beta\) to restore satisfiability.
There is a pairwise dependence \((\alpha, \beta)\) if and only if the CSP instance is satisfiable. Since \(C' \subset C\), if there are no solutions to the CSP instance then there are no solutions to the ACN, and hence there is no pairwise dependence. It is easy to see that if there is a solution \(s\) to the CSP instance, then there are two corresponded solutions to the ACN, \(s_0\) with \(\alpha = \beta = false\) and \(s_1\) with \(\alpha = \beta = true\). Since changing \(\alpha\) also changes \(\beta\), \(s_0\) and \(s_1\) are minimally different when changing \(\alpha\). Hence, if the CSP instance is satisfiable then \((\alpha, \beta)\) is a pairwise dependence. Therefore, the problem of deciding PWDR is NP-complete.

To address the scalability issue caused by constraint solving and solution enumeration, Cai and Sullivan presented a divide-and-conquer algorithm [50] to potentially reduce the number of solutions that need be processed at once. Their algorithm splits an ACN into several smaller sub-ACNs that can be solved independently, then merges the DA from each sub-ACN to derive the full DA. Although this splitting technique does indeed reduce the total number of solutions explored and improves the performance from their initial brute-force algorithm [49], it does not change the problem complexity. The number of sub-ACNs that can be decomposed depends on the quality of design rules in the system. If a system is not well modularized, the divide-and-conquer algorithm will potentially generate sub-ACNs with a large number of variables and solutions. Solving these large sub-ACNs and combining them together still exhibit exponential complexity.

The polynomial time algorithm presented in Section 4.4 does not require decomposing a large ACN and recomposing results from sub-ACNs, but only applies to ACNs with restricted forms that we introduce in Section 4.3. In Section 4.6, we compare the running time needed for our new algorithm with the time needed for the divide-and-conquer algorithm, showing that the new algorithm performs significantly better.

4.2 Formalizing UML Class Diagram to ACN

In this section, we present a formalization of a UML class diagram using an augmented constraint network, including UML classes, interfaces, and their major relations. These UML elements covered in this section are all supported by IBM Rational Rose\(^2\) and include all the relations used in the UML class diagram reverse-engineered from Apache Ant, one of the systems we used to evaluate our approaches.

\(^2\)http://www-306.ibm.com/software/awdtools/developer/rose/
4.2.1 Decisions in UML Classes and Interfaces

From the design decision perspective, each UML class consists of two design dimensions: the interface dimension and the implementation dimension. As a result, we model a class $A$ using two ACN variables: $A_{\text{interface}}$ and $A_{\text{impl}}$. Each dimension can vary, so we model the domain of each variable as having at least two values $\{\text{orig,other}\}$, where $\text{orig}$ models the current decision and $\text{other}$ models an unelaborated new choice for the decision.

The ACN translated from class $A$ is shown as below. Besides the two variables, we use a logical expression to model that the implementation of $A$ makes assumptions about its interface. We also assume that the interface of a class dominates its implementation, which is translated into a dominance relation. As a result, we translate the class $A$ into the following ACN (Figure 4.4):

<table>
<thead>
<tr>
<th>Constraint Network:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{\text{interface}} : {\text{orig,other}}$</td>
</tr>
<tr>
<td>$A_{\text{impl}} : {\text{orig,other}}$</td>
</tr>
<tr>
<td>$A_{\text{impl}} = \text{orig} \Rightarrow A_{\text{interface}} = \text{orig}$</td>
</tr>
</tbody>
</table>

Dominance Relation:

$(A_{\text{impl}}, A_{\text{interface}})$

Figure 4.4: UML Class to ACN Formalization

Since a UML interface $I$ has no implementation, we model it using only one variable, the interface variable: $I_{\text{interface}} : \{\text{orig,other}\}$.

4.2.2 Assumption Relations in UML

Given that classes and interfaces are translated into variables, we now identify the assumption and dominance relations implied in UML class diagram relations, and model them into an ACN.

Dependency, Association, Aggregation, and Composition

We consider these four relations together because from the decision-making perspective, the assumption relation among the elements of these relations are the same, and we map them into an ACN in the same way. The four diagrams in Figure 4.5 depict these relations in order.

---

3We emphasize that the $\text{interface}$ variable of an ACN represents the publicly accessible methods, fields, etc. of a class, and should not be confused with the programmatic interface construct provided by many object-oriented languages.
A dependency exists between two elements, A and B, if a change to the definition of one could result in a change to the other. This is indicated by a dashed arrow pointing from the dependent to the independent element.

An association represents a structural relationship between two classes; if there exists an association between classes A and B then objects of class A can be related to class B, such as “department offers courses.” This is indicated by a line with each end connected to a class box.

Aggregation is a variant of the “has a” or association relationship. In UML, it is graphically represented as a clear diamond shape on the containing class end of the line that connects the contained class to the containing class. Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are not. For example, “a flashlight has batteries” is an aggregation relation.

Composition is a stronger variant of the “has a” association relationship, such as “a house has rooms.” It is represented by a filled diamond shape on the containing class end of line that connect contained class to the containing class. Composition has a strong life cycle dependency between instances of the container class and instances of the contained class(es): If the container is destroyed, every instance that it contains is destroyed as well. A composition is an aggregation that must satisfy the following two constraints: (1) an object may be part of only one composite at a time, and (2) the composite object has sole responsibility for the management of all its parts.

Viewed in terms of design decisions and assumptions, the four relations shown in Figure 4.5 have the same assumption relations. A dependent class or a containing class B makes assumptions about the interface of the independent class or contained class A. In other words, the change in the interface of A will influence B’s implementation, but not vice versa. Their logical expression and dominance relation are as follows (Figure 4.6):
The diagram in Table 4.1 shows a UML class diagram for a generalization relation. The generalization relation consists of two elements, one general and one specific. Each instance of the specific element is indirectly an instance of the general element, therefore the specific element inherits the features of the general element [153]. The specific element can also provide more features than the general element.

Table 4.1: UML Generalization to ACN

<table>
<thead>
<tr>
<th>UML</th>
<th>Augmented Constraint Network</th>
</tr>
</thead>
</table>
| ![Diagram](image) | Constraint Network:  
B_impl = orig \Rightarrow A_interface = orig  
B_interface = orig \Rightarrow A_interface = orig  
B_impl = orig \Rightarrow A_impl = orig  
Dominance Relation:  
(B_impl, A_interface)  
(B_impl, A_impl)  
(B_interface, A_interface) |

In the diagram shown in Table 4.1, A is the general element and B is the specific element. Since B inherits the features of A, this means the design decision of A’s interface dominates and influences B, both its interface and its implementation, which is modeled using the first two logical expressions shown in the table.

Moreover, the implementation decision of A also influences the implementation of B. Although the changes in A will be propagated to B implicitly, and the user may not need to change B’s code, s/he should be aware of this change to avoid unwanted side effects. The third logical expression makes this relation explicit. The A_impl variable also dominates B_impl because a change in B should not force changes in A, which may be inherited by other classes. The dominance relation is thus constructed accordingly.
Interface Realization

The diagram in Table 4.2 shows the interface realization relation. In the realization relation, one element (the supplier), A, provides a specification, and another element (the client), B, provides an implementation matching said specification [153]. Similar to the generalization relation, we model the relations among the decisions of A and B using an ACN shown in Table 4.2, which is different from the generalization relation in that the parts involving $A_{\text{impl}}$ are removed, because the interface $A$ does not have an implementation.

Table 4.2: UML Realization to ACN

<table>
<thead>
<tr>
<th>UML</th>
<th>Augmented Constraint Network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Constraint Network:</td>
</tr>
<tr>
<td></td>
<td>$B_{\text{impl}} = \text{orig} \Rightarrow A_{\text{interface}} = \text{orig}$</td>
</tr>
<tr>
<td></td>
<td>$B_{\text{interface}} = \text{orig} \Rightarrow A_{\text{interface}} = \text{orig}$</td>
</tr>
<tr>
<td></td>
<td>Dominance Relation:</td>
</tr>
<tr>
<td></td>
<td>$(B_{\text{impl}}, A_{\text{interface}})$</td>
</tr>
<tr>
<td></td>
<td>$(B_{\text{interface}}, A_{\text{interface}})$</td>
</tr>
</tbody>
</table>

Owned Element Association

The diagram in Table 4.3 shows an owned element association, representing that the class $B$ (inner class) is nested within the class $A$ (outer class)—that is, $B$ is declared in the scope of the outer class $A$. An outer class and its inner class are connected by an anchor line, with an anchor icon which appears as a plus sign inside a circle on the end connected to a outer class.

In this relation, if $A$’s interface or implementation (e.g. an inner variable used by $B$) changes, $B$’s implementation may need to change. This means that the design decisions of both $A$’s interface and implementation dominate and influence $B$’s implementation, which is modeled as the ACN shown in the table.

4.3 Binary Augmented Constraint Network

Our approach is based on the observation that the majority of ACNs used to model real software systems exhibit two common characteristics that seem to be sufficient for dependency analysis. First, for dependency analysis, we mainly care about whether one decision makes assumptions on another decision, and whether changing the current decision will influence the decisions on other dimensions.
but care less about what the *current* or *changed* decisions are. As a result, the domain of a variable in an ACN often can be abstracted as having only two values: *orig* (the current selected choice) and *other* (an unelaborated future choice). The rationale is that given a changed decision, the designer first needs to know what other dimensions will be impacted, but not what the exact new choices are. Modeling a software design this way will not be sufficient to support certain property analysis as supported by other model checking techniques, such as finding compatible states of all components.

Instead, our focus is modularity and dependency analysis. Second, constraints in ACNs represent an *assumption* relation, which often can be expressed using the form $a = \alpha \Rightarrow b = \beta$ to mean that the choice for decision $a$ assumes a certain choice for decision $b$.

We call an ACN that exhibits these characteristics as a *binary augmented constraint network* (BACN, pronounced *bacon*). More precisely, a BACN is an ACN where each variable has a binary domain and each constraint is a mathematical implication of two bindings $A \Rightarrow B$. Constraints of the form $A \lor B$ and $A \leftrightarrow B$ can be trivially converted to the required form, so they are also considered valid in BACNs. According to the formalizations shown in Section 4.2, all the ACNs transformed from UML class diagrams are BACNs.

In Section 4.4, we show that, unlike the general case of ACNs, computing PWDR for BACNs is not NP-complete by presenting a polynomial time algorithm. The key to the tractability of this problem is that BACNs have both restricted constraints and domain arity. Any CSP can be transformed to one with only binary constraints [18], so only restricting the constraints does not change the problem complexity. Since each variable in a BACN has only two valid domain values, we can consider one of those values to be *true* and the other to be *false*, and the BACN becomes equivalent to a 2-CNF instance. In fact, the basis of our algorithm is to consider the bindings as literals in a 2-CNF instance, and leverage existing 2-CNF techniques.
Since the constraint network of a BACN is equivalent to a 2-CNF instance, the most obvious algorithm for computing PWDR is to solve for all satisfying solutions and construct the design automaton. At first this seems like a lucrative approach because we can find an initial satisfying solution, in linear time [15, 84], and enumerate the remaining solutions, in polynomial time per solution [85]. However, we run into a state explosion problem [60] because there are an exponential number of solutions in the worst case, making this approach infeasible for large models. In contrast, the algorithm we present in the next section does not enumerate all solutions and therefore its running time is independent of the number of solutions.

4.4 PWDR Derivation Algorithm from BACN

We use the example ACN from Figure 4.1 to illustrate our algorithm. The basic idea is to build an influence graph, whose edges are potential PWDR pairs; then use a compatibility graph [85] of the constraint network to check for the existence of transitions in the DA, in order to verify the potential PWDR pairs. Both the influence graph and compatibility graph are constructed from an implication graph, a well-known structure for solving 2-CNF problems.

The first step of our algorithm is to build an implication graph [15, 85] to model the constraints. We create two vertices in the implication graph for each variable in the constraint network: one for each domain value. For notational simplicity, we refer to the vertices (by their 2-CNF equivalents) as $v$ and $\overline{v}$ rather than by specific domain values. For each constraint $u \Rightarrow v$, we create two directed edges $(u, v)$ and $(\overline{v}, \overline{u})$. Figure 4.7 shows the implication graph for our example ACN. For example, the edges $(algo = matrix, ds = matrix)$ and $(algo = list, ds = list)$ are created for the constraint: $algo = matrix \Rightarrow ds = matrix$. We explain the significance of the dashed edges later in this section.

From the implication graph, we construct a compatibility graph, following Feder’s algorithm [85] for identifying partial solutions to a 2-CNF instance. A compatibility graph is an undirected graph, with an edge $(u, v)$ if and only if there is an edge $(\overline{u}, v)$ in the transitive closure of the implication graph (not shown). In other words, the edges in a compatibility graph represent the clauses in the transitively closed 2-CNF instance. Figure 4.8 shows the compatibility graph constructed from the implication graph in Figure 4.7.

Rather than identifying all solutions to the constraint network and then finding dependent variables from transitions between solutions, we use the compatibility graph to identify partial solutions when checking for a potential dependency between variables. Feder [85] proves that any valid vertex
cover to the compatibility graph is a valid partial solution to the 2-CNF instance, and that minimal vertex covers are full solutions. Given a vertex cover $\sigma$, if both $v$ and $\bar{v}$ are in the cover then $v$ is unassigned in the partial solution; if only $v \in \sigma$ then $v$ is assigned true, and if only $\bar{v} \in \sigma$ then $v$ is assigned false. For example, taking the top vertices in the graph in Figure 4.8 clearly forms a valid vertex cover and logically is a valid solution/design because it uses adjacency matrices for dense graphs. Therefore, if we know that changing a variable $a$ may require a change in $b$ to compensate then we can use the compatibility graph to confirm the existence of such partial solutions. We provide a detailed example of this process after defining the influence graph.

In order to account for the dominance relation and identify potential PWDR pairs, we construct an influence graph from the implication graph. To construct the influence graph, we first remove the edges from the implication graph corresponding to the dominance relations. If $u$ cannot influence
v (i.e., \((u, v) \in D\)) then we remove the edges \((u, v), (u, \bar{v}), (\bar{u}, v)\) and \((\bar{u}, \bar{v})\). The dashed edges in Figure 4.7 shows the edges that would be removed when constructing the influence graph for our example. We complete construction of the influence graph by collapsing strong components into a single vertex, because all literals in a strong component will be assigned the same value in all solutions of a 2-CNF instance \([15, 85]\). Figure 4.9 shows the influence graph constructed from the implication graph in Figure 4.7. Since \(algo = matrix\) and \(ds = matrix\) form a strong component in the implication graph, they are merged in the influence graph.

![Influence Graph](image)

**Figure 4.9: Influence Graph**

Each edge \((u, v)\) in the influence graph represents a constraint \(u \Rightarrow v\) from the ACN, so that if we change the value of \(u\) to \(true\) then we may need to change the value of \(v\) to \(true\) in order to satisfy the constraint. Therefore, we can iterate through each edge \((u, v)\) of the influence graph and use the compatibility graph to identify if \(v\) does indeed change to accommodate a change in \(u\) in any solution pair. Since we removed edges based on the dominance relation, we do not consider any edge where we are not allowed to change \(v\) in order to compensate for the change in \(u\).

We use the edge \(\{\text{density} = dense\}, \{\text{algo} = matrix, ds = matrix\}\) from Figure 4.9 to illustrate this dependence analysis process. Since \(algo = matrix\) and \(ds = matrix\) will be always have the same value, we arbitrarily pick \(algo = matrix\) for checking the compatibility graph. We are only considering \(algo\) and \(density\), so both vertices for \(ds\) are always in our vertex cover. If our potential vertex cover is \(\sigma = \{\text{algo} = matrix, \text{density} = dense, \ldots\}\) (i.e., both vertices set as \(true\)), then we have a valid vertex cover because there is no edge between \(algo = list\) and \(density = sparse\) in the compatibility graph. Changing the value of \(density\), we would need to complement \(algo\) to restore satisfiability so we look at \(\sigma’ = \{\text{algo} = list, \text{density} = sparse, \ldots\}\). Since \(\sigma’\) is also a valid vertex cover, we know that \(algo\) changes to compensate for the change in \(density\).
Sometimes changing a variable $u$ may cause the constraint to be unsatisfied but there is no way to restore consistency. In such situations, there will be no valid pair of valid vertex covers where both $u$ and $v$ are different values. This scenario often occurs when there is a self-loop in the compatibility graph and hence, a variable remains the same value in all solutions. When such a scenario occurs, we say such an edge is invalid and remove it from the influence graph since it cannot be a PWDR.

After removing the invalid edges from the influence graph, we can trivially find PWDR from the transitive closure of the resulting influence graph. The reason we take the transitive closure is because changing a variable may cause a ripple effect. For example, we consider the constraints $a \Rightarrow b$ and $b \Rightarrow c$. If all variables start as $false$, and we change $a$ to $true$, then $b$ needs to change to $true$ to satisfy the first constraint. But changing $b$ to true causes $c$ to change to $true$ to satisfy the second constraint. Since all variables in a strong component are always the same, they are all pairwise dependent. Finally, the set of edges of the transitively closed influence graph makes up the rest of the PWDR pairs.

### 4.4.1 Complexity Analysis

In this subsection, we present the running time complexity of our algorithm. For notation purposes, we let $n$ be the number of variables and $m$ be the number of constraints in the ACN. Our algorithm consists of the following steps and running times:

1. Construct implication graph: $O(n + m)$
2. Construct compatibility graph: $O(\max(n \cdot m, n^2))$
   
   (a) Transitive closure of implication graph [151]: $O(n \cdot m)$
   
   (b) Populate edges: $O(n^2)$
3. Construct influence graph: $O(n + m)$
   
   (a) Remove dominance relation edges: $O(m)$
   
   (b) Find strong components [186]: $O(n + m)$
4. Remove invalid edges: $O(m)$

   Since we only need to consider a constant number of edges to verify a valid vertex cover, we spend a constant amount of time per edge.
5. Transitive closure of influence graph: $O(n \cdot m)$
Hence, the total running time of our algorithm is \(O(\max(n \cdot m, n^2))\) polynomial. Although \(m\) can potentially be as large as \(n^2\), we expect it to be much less because a dense implication graph implies that there is very high coupling in the software design. Additionally, we construct three graphs, each with \(2n\) vertices, so our algorithm also uses polynomial space.

### 4.5 Change Impact Analysis using BACNs

In Cai and Sullivan’s previous work [49], the authors automated Parnas’s changeability analysis using a formalized change impact analysis (CIA) technique to find which and how many other variables need to be revisited if changing a decision in a particular design dimension, based on ACN and DA modeling. We recall from Section 4.1 that the function \(\delta(s_0, v, u)\) refers to the set of valid designs that result from changing the value of \(v\) to \(u\) in design \(s_0\) and that these resulting designs (1) are minimally differing from \(s_0\), and (2) do not violate the dominance relation \(D\). However, identifying these resulting designs is NP-hard in general. In this section, we describe a Turing reduction to show the complexity of CIA, and present an efficient algorithm for performing CIA on designs modeled using BACNs.

#### 4.5.1 Complexity of Impact Analysis

We describe a polynomial-time Turing reduction on how to construct a 3-coloring of an undirected graph (vertex coloring) [94], in polynomial time, using an oracle for CIA. Given an undirected graph \(G = (V', E)\), the 3-coloring problem is to construct a mapping \(\chi : V' \rightarrow \{r, g, b\}\) such that no vertex is colored/mapped to the same value as its neighbor (i.e., \(\forall \{u, v\} \in E \; \chi(u) \neq \chi(v)\)).

Our oracle for CIA takes as input: a constraint network \(N = (V, U, d, C)\), a dominance relation \(D \subseteq V \times V\), an initial solution \(s_0\), a variable \(v\), and a domain value \(u \in U\) to which the variable will be changed. We begin by constructing our constraint network from the input graph \(G = (V', E)\). For each vertex \(v \in V'\), we create a variable \(x_v\) in our constraint network and give it four possible domain values: red, green, blue, none. The value assigned to the variable in a solution corresponds to the color for the vertex. For each edge \(\{i, j\} \in E\), we create three constraints:

1. \(x_i = \text{red} \implies x_j \neq \text{red} \land x_i = \text{green} \implies x_j \neq \text{green} \land x_i = \text{blue} \implies x_j \neq \text{blue}\)
2. \(x_j = \text{red} \implies x_i \neq \text{red} \land x_j = \text{green} \implies x_i \neq \text{green} \land x_j = \text{blue} \implies x_i \neq \text{blue}\)
3. \(x_i = \text{none} \iff x_j = \text{none}\)
The first two constraints ensure that the vertices of the edge will be assigned different colors, which is a requirement of the 3-coloring problem. The third constraint ensures that if one vertex in an edge has a color, then the other vertex must also have a color. However, both vertices may be uncolored (none), allowing us to have a valid initial solution $s_0$ with all vertices uncolored.

We use an empty dominance relation $D = \emptyset$ as input to the oracle. Finally, we arbitrarily pick a vertex $v$ and ask the oracle for the impact of coloring it red, i.e., $\delta(s_0, v, \text{red})$. Since this vertex is assigned a color, the constraints force its neighbors to be assigned colors in the resulting solutions. Hence, in the solutions constructed by the oracle, all vertices in the connected component containing $v$ will be assigned a color. Only the connected component containing $v$ will be colored because it is minimally differing from the initial solution. If there are no valid coloring, then the oracle will not find any valid solutions and reply with an empty set. If there are multiple valid colorings of the connected component, the oracle will reply with all of them and we can arbitrarily pick one.

If there are multiple connected components in the graph, we can repeatedly query the oracle, using a vertex from each connected component, until the entire graph is colored. Since there are a polynomial number of connected components, the oracle is only queried a polynomial number of times. We create a constant number of variables and constraints for each vertex and edge in the graph, so the entire reduction takes polynomial time. Hence, since constructing a 3-coloring of a graph is NP-hard, computing the impact of making a change to an ACN is also NP-hard.

### 4.5.2 Algorithm

Although CIA is NP-hard in general, it can be performed in polynomial time when the design is modeled using a BACN. We use two examples to illustrate our algorithm for CIA. The basic idea is to create a vertex cover of the compatibility graph based on the initial solution, modify the cover based on the desired change, and minimally restore consistency to get the resulting solution. We begin with a simple, but abstract example, demonstrating the process of identifying and minimally restoring consistency to the vertex cover. Afterwards, we use the graph library example from Section 4.1 as a more concrete example.

**Overview**

First, we consider a hypothetical BACN model with two variables $a$ and $b$, whose compatibility graph is shown in Figure 4.10(a). We start with an initial design $s_0$ of $\{a, b\}$ and want to see the impact of changing $a$ to $\pi$. We refer to $a$ as the change source. From $s_0$, we can easily construct
a vertex cover $\sigma_0 = \{a,b\}$, denoted in Figure 4.10(a) as the vertices shaded gray. Since we want to change $a$ in the design, we modify $\sigma_0$ to become $\sigma'_0 = \{\overline{a}, b\}$. If $\sigma'_0$ were to be a valid vertex cover, then our desired change does not have any impact on the rest of the design. In that case $\delta(s_0, a, \overline{a})$ consists exclusively of the solution corresponding to the cover $\sigma'_0$. However, after making the modification though, the edge $\{a, \overline{b}\}$ is no longer covered by $\sigma'_0$ and hence, $\sigma'_0$ does not correspond to a valid design of the system. Figure 4.10(b) depicts the uncovered edge with a dashed line. In order to restore consistency to the design, we must replace $b$ with $\overline{b}$ in $\sigma'_0$.

It is simple to see that our hypothetical example demonstrates the only scenario where an edge can become uncovered from modifying the change source. More explicitly, an edge can only become uncovered when its only vertex that is in $\sigma_0$ is the change source itself. If neither vertices of the edge are the change source, then modifying the change source has no effect on the edge’s coverage. If both vertices of the edge are in $\sigma_0$, then even if one of them is the change source, the other vertex remains in $\sigma'_0$ and hence, the edge remains covered. Therefore, to minimally restore the vertex cover (and consistency to the software design), we only need to identify these problematic edges and replace them in the vertex cover. We refer to $\sigma'_0$ with the problematic edges fixed as $\sigma'$.

There are scenarios where even after fixing these edges, $\sigma'$ is still not a valid vertex cover. In these cases, changing the change source in the original design is invalid because there are no ways to restore consistency. For example, if there is an edge $e = \{a, b\}$ in Figure 4.10(a), then removing $b$ would cause $e$ to become uncovered. Additionally, if there is a self-looping edge on $a$ or $b$ then $\sigma$ would not be a valid vertex cover either. Additionally, if $b$ dominates $a$ (i.e., $(a, b) \in D$) then we are not allowed to make this change and changing $a$ in the original design $s_0$ is invalid.

Since the edges that can become uncovered are deterministically identified and they all need to be updated in order to restore consistency to the design, there is only at most one resulting
design when making a change to a BACN model. In other words, the DA for any BACN model is a deterministic finite automaton (i.e., \( \forall s_0 \in S, v \in V, u \in d(v) \left| \{ s \in \delta(s_0, v, u) \mid s \} \right| \leq 1 \)). Therefore, the impact of making a specific change are the variables that are changed to in order to restore consistency to the vertex cover.

We note that although computing CIA is very similar to determining which edges to remove from the influence graph in our PWDR algorithm from Section 4.4, it is incorrect to simply say the variables that are pairwise dependent on the change source are always in the impact scope of a change. When deciding PWDR between two variables \( u, v \) (i.e., determining whether to retain an edge in the influence graph), we check for a change in \( u \) both from changing \( v \) to \( v \) and from \( v \) to \( v \). Hence, the result of CIA is guaranteed to be a subset of the pairwise dependent variables, but not necessarily all of them.

**Example**

We use the graph library example from Section 4.1 to provide a concrete example to our CIA algorithm. Starting with an initial design \( s_0 \), with \( \text{density} = \text{dense}, \text{ds} = \text{matrix}, \text{and alg} = \text{matrix} \), we want to find the impact of changing \( \text{density} \) to \( \text{sparse} \). Or formally, we want to construct the result to \( \delta(s_0, \text{density}, \text{sparse}) \). Hence, \( \text{density} = \text{dense} \) is the change source. We can easily construct a vertex cover \( \sigma_0 = \{ \text{density} = \text{dense}, \text{ds} = \text{matrix}, \text{algo} = \text{matrix} \} \) from this solution, which consists of the top three vertices in the graph in Figure 4.8.

Since we want to change \( \text{density} \) to \( \text{sparse} \), we first remove \( \text{density} = \text{dense} \) from \( \sigma_0 \) and replace it with \( \text{density} = \text{sparse} \). Figure 4.11 shows the updated vertex cover \( \sigma'_0 \) on the compatibility graph, with the covered vertices shaded. We use dashed lines to show the edges in this graph that are not covered by \( \sigma'_0 \). Because of the uncovered edges, \( \sigma'_0 \) is not a valid vertex cover and hence, not a valid design. So we need to try to minimally restore consistency.

The edges that match our problematic scenario, discussed earlier, are \( \{ \text{ds} = \text{list}, \text{density} = \text{dense} \} \) and \( \{ \text{algo} = \text{list}, \text{density} = \text{dense} \} \). After switching both \( \text{ds} \) and \( \text{algo} \) to \( \text{list} \), the vertex cover is valid and hence the software’s design is also valid. Therefore, the resulting design is \( \{ \text{density} = \text{sparse}, \text{ds} = \text{list}, \text{algo} = \text{list} \} \). We see that this is a semantically correct design and it is indeed minimally different from the original design.
4.6 Evaluation

The purpose of our evaluation is to test the correctness and scalability of these approaches. Since software dependency structure can be represented using design structure matrices (DSMs), our evaluation focuses on the effectiveness of generating DSMs from ACNs. The DSM itself has shown to be an effective modularity analysis and visualization model [46, 49, 133, 135, 166, 183, 184]. We select two sets of subjects. The first set includes the ACNs that have been published in software engineering literature. We compare the DSMs generated from BACN with these published DSMs to test the correctness of our approach and to demonstrate the performance improvement. Since all these previously published models are not large, we also select a number of medium to large-scale real software projects as our experimental subjects. We first reverse-engineer these projects into UML class diagrams, convert these UML class diagrams into BACNs and then generate DSMs. We report the performance data to demonstrate the feasibility and scalability of both UML to ACN conversion and BACN analysis techniques. To test the accuracy of the UML to ACN conversion, we compare the BACN-generated DSMs with the DSMs generated from the source code using Lattix.

We developed a tool called JANUS to transform UML class diagrams into ACNs, and a plugin to our PARCAE framework, called BACN, to process BACNs, generate DSMs and conduct change impact analysis. Our experiments ran on a Linux server with two quad-core 1.6Ghz Intel Xeon processors and 4GB of RAM. To compare with Cai and Sullivan’s divide-and-conquer approaches, our MINOS tool reimplements the functionality of their Simon [49], and optimizes their algorithm of
by parallelizing the construction of DAs using multiple threads. We found, through experimentation, that using four threads produced the peak performance for their algorithm on our machine. The brute-force algorithm Cai and Sullivan first developed [49] simply enumerated all solutions to the ACN to build a DA, and identified PWDR from the DA transitions. Since this brute-force approach is extremely hard to scale, we do not take it into comparison. In this section, we first describe the subject software systems, and then present the experimental results, highlighting several real software systems we have investigated.

4.6.1 Subjects

We studied 70 ACNs in total, including both previously published models and ACNs derived from software projects specific for this study. The previously published ACNs (or the derived DSMs) include both small but canonical software examples widely used in software engineering literature and medium to large-scale real software systems. Concretely, for small canonical examples, we examined two ACNs representing two variations of Parnas’s keyword in context (KWIC) system [49, 156, 184], two ACNs representing AO and OO alternatives of the widely used Figure Editor (FE) example [48, 103], and a maze game illustrating design patterns [92]. These 70 ACNs are all BACNs except for the two KWIC models.

Among these previously published models, the ACNs modeling real software systems include 10 ACNs modeling variations of the WinerySearch system studied by Lopes et al. [133] and Cai and Sullivan [50], 6 ACNs abstracting from a fault tree analysis tool called Galileo [46, 182], 3 ACNs modeling the AO, OO, and DR alternatives of a networking system called HyperCast [50, 117, 132, 183], an ACN modeling a financial management system called VODKA [47], and 16 ACNs modeling 8 releases of a software product line called MobileMedia [86, 172, 173] with AO and OO alternatives for each release. The ACNs represent heterogenous real software systems. The commonality among these ACNs are that they are all derived from higher-level software artifacts than source code. The MobileMedia ACNs were automatically generated from UML component diagrams [172], all other ACNs were manually constructed from specifications and design descriptions.

Our purpose is to test whether the approaches can enable dependency analysis for large-scale models, but the largest ACN model in existing literature only has 81 variables and we do not have large-scale design-level artifacts to transform into ACNs. Similarly, to test the feasibility of transforming large-scale UML class diagrams to ACNs and the utility of such transformation, we need large-scale UML class diagrams, which are hard to obtain. To facilitate the evaluation of both
our approaches, we take real software systems of reasonable scales, reverse engineer them into UML class diagrams using our tool called Moka, derive ACNs from the UML class diagrams using our Janus tool, and then generate their DSMs using Bacn. As we discussed before, significantly more dependencies, mainly implicit or indirect dependencies, can be picked up using this technique than using traditional reverse engineering tools, such as Lattix, that extracts syntactical dependencies.

The newly studied projects we selected include the first 15 versions of Hadoop, JUnit version 3.4, SAT4J, Alloy 4, Kodkod, 8 versions of JEdit, Frostwire, Glusta, and PARCAE itself. Using PARCAE to generate a DSM modeling itself is an interesting way to assess the scalability and modularity of its own. The ACNs generated for these systems are all BACNs, and most have hundreds of variables.

4.6.2 Evaluation Questions

Given our goal of evaluating the feasibility and scalability of our approaches, we investigate a number of evaluation questions.

Using previously published ACNs, we compare against the results of Cai and Sullivan’s divide-and-conquer algorithm, and investigate the following questions:

Q1: Does our algorithm produce the same DSMs as Cai and Sullivan’s algorithm?
Q2: Does our change impact analysis generate the same results as Cai and Sullivan’s algorithm?
Q3: Does our algorithm outperform Cai and Sullivan’s algorithm, in terms of running time?

Using these newly studied projects, we assess the scalability of both Janus and Bacn against the following questions:

Q4: Does Janus process large-scale UML class diagrams with reasonable performance?
Q5: Does Bacn process the large-scale ACNs (produced by Janus) with reasonable performance?

To demonstrate the accuracy and potential utility of these approaches, we compare the UML and BACN derived DSMs with the syntactical DSMs generated from Lattix for these newly studied projects, and answer the following questions:

http://hadoop.apache.org
http://www.junit.org
http://www.sat4j.org
http://www.jedit.org
http://www.frostwire.com
Q7: What are the main differences between these two kinds of DSMs?

Q8: After using Cai and Sullivan’s algorithm [50] to decompose a large DSM into a number of smaller subsystem modules, does each sub-DSM faithfully capture all the components needed for a particular function?

4.6.3 Experimental Results with Previously Studied Systems

To answer Q1, we compared all the DSMs generated from BACNs using our algorithm with the previously published DSMs generated using Cai and Sullivan’s divide-and-conquer algorithm. The experiment shows that not all DSMs can be generated using the divide-and-conquer approach because some decomposed sub-ACNs are still too large to be solved within reasonable time without running out of memory. For all the ACNs that can be processed by both algorithms, the DSMs generated are exactly the same, providing a positive answer to the first question.

To answer Q2, we list the running time used to process the ACNs in Table 4.4. The first column shows the name of the software; the second column shows the number of variables in the ACN; the third column shows the number of constraints in the ACN; the fourth column shows the running time of Cai and Sullivan’s divide-and-conquer algorithm; and the last column shows the running time of our BACN algorithm. All algorithm running times are reported in seconds. We use “N/A” to denote when an algorithm takes longer than eight hours to run, or cannot compute PWDR due to memory exhaustion on our machine. Below, we elaborate on the results for several of the larger real systems.

HyperCast. HyperCast [132] is a scalable, self-organizing overlay system developed in Java, with roughly 50 KLOC. Hypercast has been studied in multiple software engineering work [50, 117, 183] for different purposes. Sullivan et al. [183] investigated three different designs of the system (one using object-oriented programming, one using oblivious aspect-oriented programming, and one using design rules and aspect-oriented programming). The DSMs used in this paper were manually constructed. Cai and Sullivan [50] then used ACNs to automatically generate these DSMs, and fixed several errors caused by manual construction in the previous work. Huynh et al. [117] checked the conformance between design and implementation of HyperCast.

The result shows that for all three designs of HyperCast, our algorithm was faster than the divide-and-conquer algorithm. For example, Cai and Sullivan’s algorithm took six minutes to generate the DSM for the AO design of HyperCast, but our algorithm only took less than one second.
Table 4.4: PWDR Running Time Comparison (in whole seconds)

<table>
<thead>
<tr>
<th>System</th>
<th># Vars</th>
<th># Constraints</th>
<th>Div&amp;Conq</th>
<th>BACN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Winery Locator 1</td>
<td>27</td>
<td>45</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Winery Locator 2</td>
<td>29</td>
<td>13</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Winery Locator 3</td>
<td>29</td>
<td>51</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Winery Locator 4</td>
<td>32</td>
<td>55</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>HyperCast OO</td>
<td>28</td>
<td>42</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>HyperCast AO</td>
<td>25</td>
<td>50</td>
<td>368</td>
<td>0</td>
</tr>
<tr>
<td>HyperCast DR</td>
<td>33</td>
<td>60</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Glusta</td>
<td>31</td>
<td>188</td>
<td>894</td>
<td>0</td>
</tr>
<tr>
<td>JUnit 3.4</td>
<td>62</td>
<td>269</td>
<td>6555</td>
<td>4</td>
</tr>
<tr>
<td>Mobile Media OO R1</td>
<td>29</td>
<td>64</td>
<td>6884</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media OO R2</td>
<td>32</td>
<td>74</td>
<td>67301</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media OO R3</td>
<td>38</td>
<td>90</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media OO R4</td>
<td>40</td>
<td>94</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media AO R1</td>
<td>29</td>
<td>64</td>
<td>6885</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media AO R2</td>
<td>33</td>
<td>79</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media AO R3</td>
<td>39</td>
<td>94</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Mobile Media AO R4</td>
<td>41</td>
<td>97</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>VODKA</td>
<td>161</td>
<td>185</td>
<td>N/A</td>
<td>0</td>
</tr>
</tbody>
</table>

**MobileMedia.** We evaluated our algorithm over eight releases of a software product line called MobileMedia [86]. MobileMedia contains about 3 KLOC and provides support for handling photo, music and video data on mobile devices. Each release evolves from the previous release by adding some new functionalities or by restructuring the previous release to achieve a better modularized structure. Table 4.4 reports the running times for dependency analysis on both designs of MobileMedia (one with object-oriented design and one with aspect-oriented design).

Table 4.4 shows that the divide-and-conquer algorithm not only took dramatically more time than our algorithm, but often could not even complete the computation. Their algorithm took almost 2 hours to process release one and almost 19 hours to process release two, despite the fact that only three additional variables were added to the design. On the other hand, our algorithm reduced these running times to under one second, with no noticeable increase in running time. The substantial increase in running time of their algorithm is due to the fact that the number of solutions to a constraint network can be exponential to the number of variables, and emphasizes the need for an algorithm whose performance does not depend on solution count.

After release two, the MobileMedia constraint networks have so many solutions that the divide-and-conquer algorithm can no longer enumerate all solutions with the amount of memory available on our test machine. Since BACN does not depend on the number of solutions, we are able to compute the DSMs for all eight releases of both OO and AO designs.
VODKA. We recently had the opportunity to work with a team in the Department of Computer Science at Drexel University, where we manually modeled their senior project using ACNs. The team designed and developed a web-based, service-oriented software system, VODKA Organizational Device for Keeping Assets (VODKA) [47], to standardize the financial management method among all student organizations and simplify university auditing. This system contained 154 functional requirements, 11 non-functional requirements, 20 web services, and 13 Java servlets. We modeled the entire system (including requirements, architecture/design, and test procedures) with ACNs by reading their documentation.

As Table 4.4 shows, the divide-and-conquer algorithm could not compute the DSM for VODKA: after running for almost two hours, Minos crashed with an out of memory error. We added some debugging code to Minos to report its progress when crashing and found that even with the ACN splitting technique, one of the sub-ACNs contained over 1.1 million solutions. We tried running Minos on a machine with 8GB of memory to see how long it would take to actually derive the DSM for VODKA, but after running for 49 hours and using 7.5GB of memory it was still not finished finding all the solutions. In contrast, without any splitting of ACNs, our algorithm was able to compute the DSM within one minute.

CIA Evaluation

To answer Q3, we compare the correctness and efficiency of our CIA algorithm against the existing divide-and-conquer approach of Cai and Sullivan. Using the same subjects as for evaluating our PWDR algorithm, we compared the results of our CIA algorithm against the results produced by the divide-and-conquer algorithm and compared their running times. Our approach always produces the same results as the previous divide-and-conquer approach, so we conclude that our algorithm is correct. Additionally, the running times for the CIA algorithms were nearly identical to the PWDR algorithm comparison, so we omit them from this paper.

In summary, using the 59 previously published models in existing literature, we answer affirmative to Q1 and Q3 that our algorithm computes the correct DSMs and generates the same change impact analysis results. We also answer affirmatively to Q2 that our algorithm outperforms their algorithm for all the studied models. In many cases, the memory requirements to enumerate all solutions of the constraint network prevents the divide-and-conquer algorithm from succeeding. Since our algorithm’s performance does not depend on the number of solutions, our algorithm was able to correctly derive DSMs for all the BACNs, often reducing the running time from hours to seconds.
Table 4.5: UML Formalization and PWDR Performance Results (in whole seconds)

<table>
<thead>
<tr>
<th>System</th>
<th># Class</th>
<th># Relations</th>
<th>JANUS</th>
<th># Vars</th>
<th># Constraints</th>
<th>Div&amp;Conq</th>
<th>BACN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parcae 0.0.7</td>
<td>74</td>
<td>957</td>
<td>3</td>
<td>128</td>
<td>967</td>
<td>3288</td>
<td>0</td>
</tr>
<tr>
<td>Parcae 0.1.0</td>
<td>82</td>
<td>1253</td>
<td>3</td>
<td>155</td>
<td>1424</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>SAT4J</td>
<td>147</td>
<td>2043</td>
<td>4</td>
<td>253</td>
<td>2043</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Alloy 4</td>
<td>179</td>
<td>8349</td>
<td>7</td>
<td>352</td>
<td>8685</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>Kodkod</td>
<td>247</td>
<td>4946</td>
<td>5</td>
<td>476</td>
<td>5595</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>Hadoop 0.15.1</td>
<td>703</td>
<td>11793</td>
<td>11</td>
<td>1347</td>
<td>13804</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>Apache Ant 1.7</td>
<td>700</td>
<td>11229</td>
<td>12</td>
<td>1349</td>
<td>13226</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>JEdit 4.3</td>
<td>948</td>
<td>17553</td>
<td>17</td>
<td>1832</td>
<td>19906</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>FrostWire 4.18</td>
<td>2396</td>
<td>27695</td>
<td>88</td>
<td>4456</td>
<td>32704</td>
<td>N/A</td>
<td>7</td>
</tr>
</tbody>
</table>

It is worth noting that whether or not the divide-and-conquer approach will work does not only depend on the number of variables. For example, Cai and Sullivan’s algorithm cannot generate the DSM for MobileMedia release 4 with only 40 variables, but can generate a DSM for Parcae that has 128 variables. This is due to the fact that Parcae is well modularized and its DRs, formalized as dominance relations, decompose the large ACNs into much smaller sub-ACNs.

4.6.4 Experimental Results with Newly Studied Projects

Next we describe our evaluation on ACNs derived through UML formalization.

Scalability Evaluation. To answer Q4 (the scalability of the Janus program) and Q5 (the scalability of the BACN PWDR algorithm), we reported our experimental results in Table 4.5, which shows that our approaches works for all these systems, including large-scale ones, in reasonable period of time.

The Parcae Tool. Our Parcae tool is a Java-based ACN/DSM analysis framework that implements the BACN algorithms presented in this chapter. It also provides the functionality of Cai and Sullivan’s divide and conquer [49] algorithm by leveraging our Minos tool as a library. At two separate occasions, we reverse engineered a UML class diagram from the Parcae code base and automatically derived an ACN from it.

By examining a breakdown of the running time of Cai and Sullivan’s algorithm on the first Parcae ACN that we considered (version 0.0.7 with about 10 KLOC), we clearly see the tradeoffs of the divide-and-conquer approach. Cai and Sullivan’s algorithm divided Parcae into 57 sub-ACNs. Most of these sub-ACNs had few solutions (minimum was 3) and very few sub-ACNs had a large number of solutions (maximum was 23,041), so the total time for finding all the solutions to the sub-ACNs was only about 20 seconds. On the other hand, finding the transitions of the DA
took almost half the time (24 minutes) and merging the DAs to took almost the rest of the time (20 minutes). Hence, we see that although using a divide-and-conquer approach reduces the time to solve the constraint network, it introduces additional time to merge the DAs together. Since our algorithm avoids these steps, we reduce the running time from 55 minutes to 55 seconds.

Several months later, when PARCAE version 0.1 was released, we again derived an ACN from the code base and compared the algorithm running times. However, this time Cai and Sullivan’s divide and conquer algorithm did not terminate after several days. Accidentally, we forgot to kill the process on one of our servers and stumbled upon it over a month later! After all that time, the process was still running (although the multi-level-queue process scheduler was no longer giving it much CPU time) and using almost all 8GB of RAM on the machine. In contrast, our algorithm took less than a second to compute the DSM.

**Hadoop.** One of the larger systems we used to evaluate our approach was Hadoop. Hadoop is an open source map/reduce system for distributed computing, written in the Java programming language. We reverse engineered UML class diagrams from the first twelve releases of Hadoop and automatically derived ACNs from them. Release 0.1 contained 197 classes and interfaces, and release 0.15.1 contained 704 classes and interfaces. Due to the number of solutions to the constraint network, the divide-and-conquer algorithm could not produce a DSM on our test machine for most of the releases. Our algorithm, for example, took less than one second to process version 0.1 and around two seconds to process version 0.2.

**Accuracy Evaluation**

We use Apache Ant as an example to illustrate Q7 and Q8, which concern the accuracy of our approaches. Figure 4.12 shows a small portion of the UML class diagram we derived. It is obvious that a UML diagram of this size becomes very difficult to understand. We decompose the diagram into much smaller modules so that each can be comprehended independently, and demonstrate the correctness and potential utility of our approaches by examining these subsystems.

The UML model, as the input of our JANUS program, contains 626 classes and interfaces with over 3000 inter-component relations. The resulting ACN contains 1349 variables and about 13000 constraints. Our JANUS tool converts this UML class diagram into an ACN in approximately 12 seconds, and it takes PARCAE a few seconds to decompose the big ACN into 435 sub-ACNs.
Figure 4.12: Apache Ant UML Diagram

To investigate Q7 and Q8, we compare the BACN-derived DSM with the DSM reverse-engineered by Lattix. In a Lattix DSM, the variables are the classes, functions, and variables, and the dependencies are direct, syntactic references, including the usage, inheritance, and invocation relations among these constructs. By default, Lattix uses classes and packages as units when automatically deriving a DSM from a source code repository. The comparison shows that the dependency captured in a BACN-derived DSM is the superset of the dependencies captured by the corresponding Lattix DSM, and that a BACN-derived DSM has many more dependencies than that of a corresponding Lattix DSMs.

To examine the differences and accuracy, we use a divide-and-conquer approach to provide a more efficient way to make detailed comparison. As shown in Cai and Sullivan’s prior work [50], after decomposing an ACN into a set of smaller sub-ACNs, each sub-ACN (and hence the derived sub-DSM) contains all the decisions needed to implement a particular feature of the design. Given such a sub-DSM, we find the sub-module for the same feature in the corresponding Lattix DSM by moving together all the classes the function depends on. After that, we compare the two kinds of sub-DSMs to see if there are any discrepancies. If there are, we further analyze the reason behind: do the dependencies provide additional insights? Which DSM faithfully captures all the dimensions needed for developing and maintaining the feature?
We perform this comparison on a dozen pairs of sub-DSMs of varying sizes from Ant. We find that a subsystem decomposed from a BACN-derived DSM indeed captures all and only the decisions and dependencies needed for a function, and our DSMs derived from the UML model provide additional insights than the corresponding source code DSMs. We use one of decomposed subsystem to demonstrate the differences below.

We use PARCAE to decompose the Apache Ant BACN model into 435 sub-ACN models. Each sub-ACN has 2 to 86 variables, representing a feature whose class implementation dimension is one of the minimal elements in the condensation graph. Accordingly, we use the name of the minimal element as the name of the sub-ACN. For example, a sub-ACN named `org.apache.tools.ant.types.resources.selectors.Name_impl` means that this sub-ACN concerns the implementation of the Name class. We derive the DSM models for each of the decomposed modules, and compare them with the sub-DSMs of Lattix by identifying a block representing the same feature.

We found two types of discrepancies between BACN and Lattix sub-DSMs: (1) the BACN sub-DSMs reveal indirect and implicit dependencies; (2) the BACN DSMs lack the dependencies irrelevant to the function it models. In other words, a subsystem module decomposed from BACN captures exactly the set of dimensions needed for the function.

We use one of the smaller modules decomposed by our technique for illustration, a module regarding the implementation of the `org.apache.tools.ant.types.resources.selectors.Name` class. Figure 4.13(a) shows the BACN sub-DSM of this module generated by PARCAE, and Figure 4.13(b) shows the corresponding source sub-DSM extracted from Lattix.

Figure 4.13: Maze Game Name Module DSMs

(a) Our Generated ACN for Name Module (b) Lattix Generated ACN for Name Module

(1) Figure 4.13(a) has one more variable (i.e., row in the matrix) than Figure 4.13(b). This is because our framework models a class as having two dimensions, interface and implementation, while a Lattix DSM doesn’t. So the last two elements in Figure 4.13(a) correspond to Name in Figure 4.13(b).
(2) Three dependencies exist in Figure 4.13(a), but not Figure 4.13(b), which are marked using dark background and white font. The discrepancies are due to the fact that our UML-derived DSMs explicitly reveal implicit and indirect dependencies—for example, caused by inheritance.

First, the BACN sub-DSM has dependencies from Name_impl to DataType_interface and ProjectComponent_interface while this dependencies are not captured by the Lattix sub-DSM. The reason is that the Resource class inherits from the DataType class, and if the DataType class’s interface changes then the Resource class’s interface would also automatically change. Since the implementation of the Name class uses the Resource class, it is depending on the Resource class’s interface. Hence if the interface of DataType changes, it would change the interface of Resource and potentially require a change to the implementation of the Name class. Therefore, although there are no explicit dependencies from Name to DataType, the inheritance hierarchy of Resource implies this implicit dependency. Name_impl depends on ProjectComponent_interface for the same reason.

Second, the DSM generated by our approach has a dependency from Resource_interface to ProjectComponent_interface but this dependency is missing from the Lattix DSM. Since the Resource class derives from DataType, which derives from ProjectComponent, part of the interface of Resource is defined in ProjectComponent. If the interface of ProjectComponent changes, then the interface for Resource also automatically changes. Hence, there is an implied dependency from the interface of Resource to ProjectComponent.

Most of the discrepancies we found in our experiment are due to these kinds of indirect and implicit relations among classes, which are not captured by Lattix DSMs. It is, however, important to expose these dependencies to designers explicitly to anticipate the ripple effect of changing parts of the system. Next, we discuss the other type of discrepancy we found in the comparison.

(3) There exists a dependency in the Lattix source sub-DSM in Figure 4.13(b) that does not exist in our BACN sub-DSM in Figure 4.13(a). The reason is that our approach splits each class into two separate design decisions (interface and implementation) while Lattix does not. The dependency expressed in the Lattix DSM is that the SelectorUtils class uses the Resource class in its implementation. But this module is concerned with the implementation of the Name class so this dependency is irrelevant to the implementer of Name in this context. The implementer of the Name class does not need to deal with potential changes to the implementation of the SelectorUtils class due to changes in the Resource class interface. All the implementer of the Name class is concerned with regarding the SelectorUtils class is its interface. The implementation is encapsulated and can change without affecting the Name class. In addition, this dependency is not missing from the
complete BACN DSM. In module 349 of our decomposed ACNs, it does indeed show that the implementation of `SelectorUtils` depends on the interface of `Resource` because that module focuses on the implementation task for `SelectorUtils`.

In summary, our sub-DSMs generated by `Janus` and `Bacn` indeed aggregate exactly the set of decisions needed for realizing a feature, making explicit both indirect and implicit dependencies embodied in a UML class diagram. The separation of interface and implementation dimensions allows us to filter out superfluous dependencies that implementers of specific parts of a system do not need to be concerned with.

4.7 Discussion

In this section, we discuss possible threats to validity and possible future work.

We only present the formalization of the major UML class diagrams elements and relations. A relation can have multiple variations, such as multiplicity in an association relation, higher order associations with more than two ends, and an association can have role names, ownership indicators, multiplicity, and visibility adorned in the end. Since our framework only concerns the assumption relations among these components, we only show a uni-directional, navigable association in the second diagram of Figure 4.5, and these variations do not change the way how their ACN models are generated. The `object constraint language` (OCL) [152] also expresses constraints and relations in UML diagrams, but it does not model dominance relation and cluster set and does not provide the functionalities we desire, such as module decomposition and DSM derivation.

Our results would be more theoretically significant if we can show that most ACNs are BACNs. One possible way is to perform the running time comparison using input ACNs randomly generated from a uniform distribution [74, 89]. We chose not to evaluate our algorithm this way but rather to use ACNs in existing literatures or transformed from real software systems because ACNs that the algorithm will encounter in practice may not be uniformly distributed in the problem space. Although an algorithm may have high probability of being efficient for any randomly selected input, the inputs it will be given in practice may be exclusively from the subset where it performs poorly.

In other words, although it is possible that the probability of randomly picking an ACN that is also BACN (from all possible constraint networks) could be very low, if by using an ACN, the user just needs to analyze current and future status of each design dimension, it is highly possible that an ACN constructed for this purpose is a BACN. Our evaluation shows that real ACNs are from a
non-uniform distribution, and all the ACNs we found had relatively few constraints (compared to the maximum number of possible constraints). These ACNs, modeling multiple heterogenous real software designs, were the inputs for our evaluation.

Despite the benefits of our algorithm over that of Cai and Sullivan’s divide-and-conquer algorithm [50], our algorithm only processes BACNs and not all ACNs. On the other hand, their algorithm can work on the general case of ACNs, at the expense of both time and memory. Another benefit of the divide-and-conquer approach is that the decomposed sub-ACNs represent sub-systems, each with all the decisions necessary to implement a feature. Decomposing and dependency generation are two orthogonal problems. We can still use the divide-and-conquer method to decompose and investigate sub-systems, and use our algorithm to extract the dependency structure for sub-systems or the whole system without the need to solve and recombine the results from sub-systems. The influence graph we used as part of the BACN algorithm can generate similar subsets of an ACN. We will further investigate this dimension in the future. Finding more efficient algorithms for the general case of ACNs is also our future work.

Our future work also includes further optimizing these algorithms. Many optimization techniques may be performed to improve its performance. A possible direction for optimization is to use multiple threads to parallelize parts of the algorithm, or avoid the construction of the three graphs. One area we are exploring is to parallelize the computation of transitive closures [26]. Investigating the benefits of such parallelization is a potential future work.

Besides running time, we are also investigating the reduction of memory requirements of our algorithm. In particular, part of our future work is to determine if all three graphs are required in the algorithm, or if a single structure is sufficient to computing PWDR.

As we have discussed, our JANUS approach identifies indirect and implicit dependencies that are not revealed by traditional reverse-engineering-based dependency extraction methods. Our hypothesis is that these extra dependencies will improve our ability to estimate change impact and communication requirements. Testing this hypothesis is our ongoing work. This chapter only focuses on the scalability issue of automatically deriving dependency structures from formal models. Chapter 5 considers the use of these dependencies for identifying coordination needs among developers.

Although our algorithm shows the possibility of leveraging large-scale formal models in dependency analysis, deriving useful dependencies from existing and heterogeneous software artifacts, and automatically transforming and combining them into BACNs remain to be challenging future work. In this chapter, we only considered converting UML class diagrams to ACN. Sethi [172] presented
a similar technique for formalizing UML components into ACN models. The ACNs derived using his approach are also BACN models. Automatically converting other UML diagrams types (e.g. sequence diagrams) into ACNs is part of our future work.

UML diagrams only embody part of design decisions of the design space. There could be possible decisions that are not representable by any UML diagrams, such as power consumption or other environmental conditions [184]. As a result, the ACN model derived from UML models can be incomplete. On the other hand, an ACN model is expandable. New design decisions and constraints can always be added.

Since the DSMs that Lattix generates miss indirect dependencies, a possible method to consider to improve those DSMs would be to take the transitive closure. In fact, in our previous work [117], we compared transitively closed source code DSMs against ACN generated DSMs, and found that the transitively closed DSMs contained redundant and false dependencies.

4.8 Related Work

Our work is related to Alloy [118], a lightweight modeling language used to express and enforce constraints among software components. The purpose of Alloy to ensure the correctness and consistency among object states, which is different from our ACN models that aim to model decisions and the assumptions among them. Cai and Sullivan's Simon [46, 49] tool leveraged Alloy as its underlying constraint solver. Our Minos tool employs Kodkod [188], the same constraint solver that is also used by the latest versions of Alloy, when solving a constraint network is needed.

Unlike many other formal modeling languages, ACNs are designed specifically for analysis of dependencies and software modular structure, based on assumptions among design decisions. For example, while Rosetta [7] also allows the modeling of assumptions and decision domains, it focuses on checking such properties as security, power consumption, and timing constraints. In addition to the differences in terms of purposes, ACNs also differ in that ACNs formalize the notion of design rules from Baldwin and Clark's modularity theory [22], and enable automated DSM derivation, changeability analysis, design modularity and stability measurement [48, 49, 172, 173, 183, 184].

General PWDR algorithms suffer from a state explosion problem [60], similar to many model checking algorithms. The ACNs we evaluated often could not be processed by Cai and Sullivan’s divide-and-conquer algorithm due to the exceedingly large demand for memory to enumerate all solutions to the constraint network. Approaches such as binary-decision diagrams (BDD) [43, 45],
and *partial order reductions* [61] offer promise in addressing state explosion in model checking. Research with BDD-based techniques have allowed model checking of systems with as many as $10^{120}$ states [44]. Applying similar techniques to computing PWDR is our future work.

There is active research on optimization of SAT solvers and constraint satisfiability [126,139,188]. While the current Cai and Sullivan algorithm would benefit from such work, there is still an issue of state explosion and storing all solutions. Our experience in using Minos is that with an efficient SAT solver, the time to compute the DA from the set of solutions is often longer than the time to enumerate the solutions. Section 4.6 discussed this observation during our evaluation.

### 4.9 Conclusion

In this chapter, we defined a restricted, but representative, form of ACN models, called BACN. We showed that, unlike the general ACN model, computing PWDR and performing CIA on BACN models can be performed in polynomial time. We also described an approach to automatically convert UML class diagrams into BACN models. To evaluate our BACN algorithms, we generated DSMs from ACNs modeling heterogenous software, and compared the results against an existing divide-and-conquer algorithm. To evaluate the UML conversion approach, we reverse engineered UML class diagrams from large-scale software projects and compared the BACN-generated DSMs against DSMs reverse engineered by the Lattix tool. The results demonstrate the potential of conducting formal and automated modularity analysis on large-scale software designs. Hence, due to (1) its formalization of the design rule theory [22], (2) its scalability of dependency analysis, and (3) the ability to derive it from existing UML models; we use the BACN model as the underlying design/dependency model for the approaches presented in Chapters 5 and 7.
5. Identifying and Assessing Concurrent Work

In today’s large-scale, distributed software development projects, it is increasingly crucial to maximize the level of concurrency among development tasks, and at the same time avoid incurring huge coordination overheads among development teams tasked with concurrent work.

It has been long recognized that software modularization plays a critical role in streamlining project coordination, as the need for coordination among developers is closely related to the dependencies between the system modules [65, 156]. Numerous researchers have explored the interplay of coordination—in particular in the form of personal communication—and modularization for large-scale software systems [10, 54, 70, 71, 107]. Still, prevailing models of design, such as UML, are not equipped with formal means to provide software project managers with explicit guidance on how development tasks can be constructed, partitioned, and assigned to maximize the parallelization of developers’ work, based on the dependency relations among the software modules they describe.

Parnas’s information hiding principle [156] and Baldwin and Clark’s design rule theory [22] provide key (although non-operational) insights about the relation between software modularization and task assignment. Parnas defines a module as an independent task assignment, a concept that is not equivalent to the conventional understanding of modules as structural constructs (e.g. functions, classes). As we described in Chapter 2, Baldwin and Clark define design rules as stable decisions that decouple otherwise coupled decisions. Example design rules include abstract interfaces, application programming interfaces (APIs), etc. The more subordinate decisions that depend on a design rule, the more influential it is, and the more important it is to keep it stable. Identifying design rules and their impact scopes is not trivial in large-scale systems.

In this chapter, we present an approach to automatically cluster a software dependency structure into a design rule hierarchy (DRH) model that manifests Parnas’s and Baldwin and Clark’s definition of module and design rule. In this hierarchy, the decisions within the top layer^1 of this hierarchy are the most influential design rules, which dominate the rest of the system, and need to be kept stable. The decisions within subsequent layers assume design decisions in previous layers. The design decisions within each layer are clustered into modules. Since modules within the same layer are independent from each other, they become candidates for concurrent implementation.

\[^1\text{We emphasize that a layer in the design rule hierarchy is not the same as in the layered architectural style.}\]
We claim that this hierarchy, populated with sufficient dependency relations, can shed light on the interplay between software structure, task parallelism, and developers’ coordination needs. Concretely, the DRH predicts that developers working on different modules within the same layer do not have communication requirements [54]; whereas dependencies between modules located in different layers, or within the same module, create communication requirements among developers working in those contexts.

The accuracy of the DRH predictions on coordination requirements fundamentally depends on the quality of the underlying model of software dependency. Cataldo et al. [52] show that, for instance, syntactical dependencies extracted from source code are not as effective as evolutionary dependencies in terms of individuating coordination requirements. In Chapter 4 we described how to automatically derive pairwise dependency relations (PWDR) from the formal augmented constraint network (ACN) [46,49]. An ACN expresses design decisions as variables, models how those decisions make assumptions about each other using logical constraints, and complements the constraint network with a dominance relation that formalizes the concept of design rules.

In this paper, we use the pairwise dependency relation derived from an ACN as the basis to form the design rule hierarchy. Its efficiency for coordination requirement analysis is discussed in the evaluation section. The PWDR can be used to automatically derive a design structure matrix (DSM) [180] with rigorous semantics [46,49,50]. We developed several tools to support ACN modeling and a number of modularity analyses, including automatic DSM derivation. These tools are further described in Chapter 8.

We evaluate the effectiveness of the design rule hierarchy, in terms of identifying parallel task assignments and manifesting the impact scope of design rules, by postulating the following hypotheses: first, developers working on different modules within the same DRH layer engage in technical communication substantially less than other groups of developers. Second, on the contrary, the need for technical communication is particularly strong for those developers that work on modules that have cross-layer dependencies. Third, the position of a design rule in the hierarchy reflects its importance in the system, with the decisions at the top level being the most influential on the overall system design.

To test the first two hypotheses, and since we did not have an opportunity to study an ongoing project and predict the flow of “live” technical communication among developers, we organize our evaluation around the mining of publicly available archives and repositories of two open source
projects, Apache Ant\textsuperscript{2} version 1.6.5 and Eclipse JDT\textsuperscript{3} version 3.2. For each project, we reverse engineer its code base into a UML class diagram and formalize the UML model into an ACN model, following the heuristics described in Section 4.2. From the ACN model, a DRH is derived to predict coordination structure. As discussed in Chapter 4, ACN modeling identifies indirect and implicit dependencies that are not explicitly available from UML models. For example, we derive 12,596 dependencies from the Apache Ant ACN, while Lattix only shows 1,700 dependencies from the code base. Furthermore, we mine the developers’ mailing list and bug tracking system, and the version control repository of the project during the period leading to the following release to infer concurrent development tasks and related communications between developers.

Through statistical analysis, the first two hypotheses are shown to be true: developers working on different modules within the same layer of the DRH communicate significantly less than other groups; and, the majority of interpersonal communications happens instead between developers working on dependent modules located in different layers. To test the third hypothesis, we similarly automatically generate a DRH for a tool we developed, called Parcae. Since we developed Parcae, we can easily determine whether the hierarchy correctly reveals the importance of its design rules. Again, our hypothesis is supported; for example, the top layer of the Parcae DRH contains the most influential decisions of the system.

In the rest of the chapter we illustrate our method and our experiments as follows: Section 5.1 briefly reviews related work. Section 5.2 describes our clustering approach. Section 5.3 presents our evaluation experiments. Section 5.4 discusses threats and future work, and Section 5.5 summarizes our contributions.

5.1 Related Work

Organization of software as a layered, hierarchical structure has been advocated for many years. Dijkstra [72] proposed this idea with the restriction that layers could only communicate with adjacent layers. Today, the layered style is popular in software architecture [24,174]. A major difference with our DRH is that the modules in these architectures are often defined based on classes, or other programmatic constructs whereas the modules in our approach are intended to indicate independent task assignments.

\textsuperscript{2}http://ant.apache.org
\textsuperscript{3}http://www.eclipse.org/jdt/
Our design rule hierarchy is different also from other well-known hierarchical structures, such as the “uses” hierarchy defined by Parnas [157] and hierarchies defined according to conceptual domains [24, 72]. For example, if a decision $A$ in a GUI layer is the only decision to depend on a decision $B$ in the business layer, then our DR hierarchy algorithm will aggregate $A$ and $B$ into a single module because these decisions can and should be made and changed together.

The DR hierarchy represents a specific clustering for a software design. Traditional software clustering techniques (e.g. Hutchens and Basili [116], Sarkar et al. [167], Arch [169], Bunch [137, 144, 145]) often cluster designs based on coupling and cohesion measures. Our DRH clustering algorithm does not explicitly consider coupling and cohesion (although cohesion is implicitly considered through the identification of strongly-connected components). Rather, our approach clusters a design by identifying design rules, in order to reveal clusters that can be developed concurrently.

Another related area is project management and scheduling. For example, Jalote and Jain [119] presented a scheduling algorithm that considers task dependencies, available skills, and resource constraints but their approach expects a task graph as an input. Our approach complements their approach in that we can use the DRH as an input task graph to their algorithm. We then can use their algorithm to elaborate on our hierarchy’s task assignments while considering other issues such as resource constraints.

To determine the order of class integration when testing object-oriented software, Briand et al. [38] discussed several graph-based algorithms that identify strongly-connected components (SCCs) and perform topological sorting to obtain a partial ordering. Heuristics are presented to break cycles within each SCC and reduce the effort of creating stubs for tests. Although our approach also identifies SCCs in a graph (for constructing a condensation graph), we do not need to break cycles because cycles in our graph represent cohesion and we identify modules based on this cohesion. In addition, the modules identified by our algorithm do not directly correspond to the SCCs of a graph.

The Lattix tool automatically reverse-engineers a DSM from a code base and provides several partitioning algorithms to identify modules and reorder the DSM into block triangular form. We tried to cluster the maze game DSM to see if these algorithms can generate a clustering similar to our DR hierarchy. The results show that even if we manually feed Lattix with the dependencies derived from the ACN (because Lattix itself does not detect indirect or implicit dependencies between variables), the partitioning algorithms either generate modules that are not cohesive—that is, contain classes with dramatically different semantics—or do not correctly reveal the order of design rules.
Our approach presents unique traits, which derive from the fact that other object-oriented modeling methods do not separate the interface and implementation dimensions for the purpose of task assignment, nor identify and use indirect and implicit dependencies. Chapter 4 already described some of these benefits.

5.2 Approach

In this section, we first present an overview of the design rule hierarchy clustering algorithm using a simple example. Then, we formally define the DRH structure and the algorithm to derive the DRH from a software dependency graph.

5.2.1 Overview

We use a maze game example to introduce the design rule hierarchy that reveals design rules and independent modules, and to illustrate how the hierarchy is derived from an ACN. Figure 5.1 shows a UML class diagram of a small system for building a maze in a computer game; this system uses the abstract factory pattern, as described by Gamma et al. [92]. A maze is defined as a set of rooms; a room knows its neighbors, such as a wall, or a door to another room. The class MapSite is the common abstract class for all the components of the maze. The UML class diagram shows two variations of the maze game supported by the abstract factory pattern: (1) an enchanted maze game (EnchantedMazeFactory) with a door that can only be opened and locked with a spell (DoorNeedingSpell), and a room that has a magic key (EnchantedRoom); (2) a bombed maze game (BombedMazeFactory) that contains a room with a bomb set in it (RoomWithABomb) and a wall that can be damaged if a bomb goes off (BombedWall).

From a UML class diagram such as this, it is not easy to determine what modules can be developed in parallel development tasks. If a team is assigned the task of implementing the enchanted maze game, they will have first of all to examine the diagram to determine all the classes that must be developed. In addition, they must be aware of all other classes, such as MapSite, with which each enchanted maze game component must interact. These classes may be designed by other colleagues, creating dependencies between tasks. In addition, UML models can scale up to a point in which tracing all relations among the classes to determine these dependencies becomes difficult [77].
Design Rule Hierarchy

Figure 5.2 shows a design structure matrix (DSM) of the maze game example, derived from the UML class diagram shown in Figure 5.1. The DSM shown in Figure 5.2 is clustered into a four-layer design rule hierarchy (DRH). Each module within a layer depends only on the decisions within the layers to the left of it in the DSM. Because there are no dependencies between the modules within each layer, those modules can be assigned as concurrent tasks.

In Figure 5.2, the four outer groupings show the layers in which tasks can be completed in parallel. The first layer identifies design rules that are most influential and should remain stable. In Figure 5.2, the first layer consists of the variables Maze\_interface and MapSite\_interface. Changing these design rules can have drastic effects on a system. For example, changing a public method signature in the MapSite class may require changes to almost all parts of the software (as shown by the numerous marks under column 2).

The second layer, from row 3 to row 6, contains decisions that only depend on the top layer decisions. Similarly, the third layer, from row 7 to row 13, contains decisions that make assumptions about decisions made in the first two layers. Each inner cluster within a layer, called a module, contains decisions that should be made together, such as the MazeFactory\_interface (row 7)
Figure 5.2: Maze Game DSM

and MazeFactory_impl (row 8) decisions. The decisions made in an inner cluster can be made at the same time with other inner cluster decisions within the same layers. For example, although MazeFactory_interface and DoorNeedingSpell_interface do not belong to the same layer of an inheritance hierarchy, they are in the same DR hierarchy layer because once the DRs in the previous layer are determined, these decisions can be developed concurrently.

The last layer of the hierarchy identifies independent modules, consistent with Parnas’s definition. Not only can these modules be designed and developed concurrently with each other, but they can also be swapped out for different implementations without affecting the rest of the system. For example, although Wall_impl is a parent class, it does not decouple other modules, and is only used by the BombedWall_impl. As a result, Wall_impl is not a design rule in the current system, and the developers of these two classes can work together for a better Wall implementation without worrying about unwanted side effects.

**DR Hierarchy Clustering**

To compute a DR hierarchy, we first need to identify all decisions needed for each task. Then we identify which of these decisions are shared by other tasks, as well as which can be taken independently and concurrently.
Identify Decisions Needed by a Task. The first step leverages Cai and Sullivan’s algorithm [50] that decomposes an ACN into a set of sub-ACNs. A sub-ACN contains the set of decisions needed to realize a particular feature of the design. We refer to this algorithm as the DECOMPose-MODULES algorithm. Its basic idea is to model the constraint network as a directed graph. In this graph, each vertex represents an ACN variable. If two variables $u$ and $v$ appear in the same constraint expression then two edges $(u, v)$ and $(v, u)$ are added to the graph. Then edges of the graph are removed using the dominance relation of the ACN: if $u$ cannot influence $v$, then the edge $(u, v)$ is removed from the graph.

After that, the condensation graph is derived from the constructed directed graph. According to its definition, each vertex of a condensation graph represents a strongly-connected component, comprising a set of variables of the original directed graph. A condensation graph is known to be a directed acyclic graph (DAG). Figure 5.3 shows a partial maze game condensation graph generated from the maze game ACN. Note that the edge directions in the graph may seem counter-intuitive. This is because the edges do not represent the direction of dependence but rather the direction of influence. In other words, if an edge $(u, v)$ exists in the graph then a change in the decision of $u$ may potentially influence the decision of $v$.

To generate sub-ACNs, all the variables along a path ending with the same minimal element (i.e., vertex with no outgoing edges) are put into a sub-ACN with the relevant subset of constraints and dominance relation. As a result, the ACN is decomposed into a set of sub-ACNs that can be processed individually. The number of sub-ACNs is equal to the number of minimal elements of the condensation graph.

Cai and Sullivan [50] observed that each minimal element of the condensation graph represents a feature of the design, and all the chains ending with a minimal element contain all the decisions needed to realize that feature. For example, Figure 5.3 shows that one of the sub-ACNs will contain the variables BombedWall_impl, BombedWall_interface, Wall_interface, and Map-Site_interface. This sub-ACN contains all the decisions needed to implement BombedWall_impl.

Simply identifying all the decisions needed for a feature does not guarantee that the tasks can be implemented or changed independently because some of the decisions may be shared by other tasks. For example, the BombedWall_impl sub-ACN contains decisions, such as MapSite_interface, that overlap with other tasks in the condensation graph. We differentiate the sub-ACNs in Figure 5.3 with different line styles and label them as $s_1, s_2, s_3$ for illustration.
Identify Shared Decisions. Our goal is to identify a hierarchy from the condensation graph, which is generated as a by-product of the \textsc{Decompose-Modules} algorithm, and further decompose these sub-ACNs into independent tasks. We call this hierarchy the design rule hierarchy because the structure is determined by the design rules, formalized as the dominance relation of the ACN.

Intuitively, our algorithm identifies each region of intersection in the condensation graph and separates each into an individual group. For example, there are two regions of intersection in Figure 5.3. In the intersection $s_1 \cap s_2 \cap s_3$, \texttt{MapSite\_interface} is shared by all three sub-ACNs so we create a new module exclusively consisting of it. Since the variable in this intersection is shared by all the sub-ACNs, its development must be completed first. The other region of intersection, between $s_2$ and $s_3$, contains \texttt{Room\_interface} so we create a new module consisting of it. Similarly as before, since $s_2$ and $s_3$ share this variable, it must be completed before the design of the concrete
Room classes. Since $s_2$ and $s_3$ no longer share any variables, they are identified as concurrent tasks. Figure 5.4 shows the hierarchy our algorithm produces from Figure 5.3 after aggregating these intersections into separate modules.

Since the resulting graph after applying this algorithm is directed acyclic, we can apply a modified breadth first search of the vertices and get a partial ordering. In other words, if we complete the tasks in Figure 5.4 from the top down, then we will not need to wait for dependencies to be completed. Figure 5.2 shows the DSM of the full maze game example in which the variables are clustered according to the result of our algorithm. All the dependencies in the DSM are either below the diagonal or within the inner groups. Sangal et al. [166] refer to this as being in block triangular form. As we will prove in Section 5.2.2, our algorithm always produces a clustering that puts a DSM in block triangular form.

5.2.2 Formalization

In this section, we formally define the design rule hierarchy, present the clustering algorithm, prove its correctness, and analyze its complexity. A DR hierarchy is a directed acyclic graph (DAG) where each vertex models a task; each task is defined as a set of design decisions that should be made together. Edges in the graph model an assumption relation: an edge $(u, v)$ models that the decision $v$ assumes decision $u$. Based on ACN modeling, a change in the choice for $u$ may cause a change in the choice for $v$. The layers within the DR hierarchy obey the following rules:

- Layer 0 is the set of tasks that assume no other decisions. We refer to this as the top layer of the hierarchy.
- Layer $i$ ($i \geq 1$) is the set of all tasks that assume at least one decision in level $i - 1$ and assume no decisions at a layer higher than $i - 1$. Within any layer, no task assumes any decisions in another task of the same layer. Hence, the tasks within the same layer can be completed independently and in parallel.
- The highest (bottom) layer is the set of independent modules. No decisions outside of these modules make assumption about any decisions within these modules.

DR Hierarchy Algorithm

Our DRH algorithm starts with identifying all the decisions needed for each feature, by first decomposing an ACN into a set of sub-ACNs using the DECOMPOSE-MODULES algorithm [46, 50].
As discussed in Section 5.2, the key structure of the DECOMPOSE-MODULES algorithm is the condensation graph $C$ derived from the directed graph $G$ representing the constraint network and the dominance relation pairs of the ACN. The output of the DECOMPOSE-MODULES algorithm includes a set of sub-ACNs, $S$. Our DRH algorithm takes both $C$ and $S$ as input, and outputs a clustering that conforms to the formal definition of DR hierarchy. Figure 5.5 lists our algorithm pseudocode.

### DECOMPOSE-HIER($C, S$)
1. create hash table $T$
2. for each $v \in V[C]$
   3. $\text{do } region \leftarrow 0$
   4. for each $s \in S$
      5. $\text{do } region \leftarrow region \times 2$
      6. if $v \in s$
         7. then $region \leftarrow region + 1$
   8. $T[region] \leftarrow T[region] \cup \{v\}$
9. create graph $H$
10. $V[H] \leftarrow \text{items in } T$
11. for each $(u, v) \in V[H] \times V[H]$
12. $\text{do if } \{(i, j) \in E[C] \mid i \in u \land j \in v\} \neq \emptyset$
13. then $E[H] \leftarrow E[H] \cup \{(u, v)\}$
14. return $H$

**Figure 5.5:** DR Hierarchy Algorithm Pseudocode

As stated in Section 5.2, intuitively, our algorithm separates each region of intersection of the sub-ACNs into a separate group. To identify which region a vertex of $C$ belongs to, we assign to each vertex an identifying bit-field of $|S|$ bits (represented by the integer $region$ in the pseudocode). For each sub-ACN $s_i \in S$, if a vertex is contained in that sub-ACN then the $i$-th bit of its bit-field will be set to 1. For example, in Figure 5.3, `Room Interface` is contained in sub-ACNs $s_2$ and $s_3$ so it would have an identifying bit-field of 110 and `Wall Interface` is only contained in sub-ACN $s_1$ so it would have a bit-field of 001.

After identifying regions, we build a new graph $H$ in which each vertex represents a region. The final for-loop in the pseudocode populates the edges of $H$ based on edges in the condensation graph $C$. The graph $H$ becomes our DRH model. To derive the DSM clustering from $H$, we first isolate the independent modules, then perform a modified breadth-first search (BFS) on the graph. The traversal is modified so that a vertex is not explored until all its incoming neighbors have been explored. While performing a topological sort also would create a partial ordering, the layered structure would not be explicitly identified.
Proof of Correctness

To show that our algorithm correctly finds a hierarchical structure that obeys the three rules defined above, we prove Theorem 1 below. The basic idea is that, since the input condensation graph $C$ does not contain any cycles, the only way a cycle can be formed in the resulting graph $H$ is through the grouping of vertices in $C$. Below, we show, by contradiction, that our algorithm does not form cycles from the vertices of $C$. To simplify this proof, we first prove Lemmas 1, 2, and 3.

**Lemma 1.** If vertices $u$ and $v$ from the condensation graph $C$ are grouped together in the result graph $H$, then for all sub-ACNs $s \in S$ : $u \in s \iff v \in s$.

**Proof.** The proof to this lemma can intuitively be seen from the construction of the result graph. Let $r(u)$ and $r(v)$ be the bitsets constructed for $u$ and $v$, respectively. We assume by contradiction that there exists a sub-ACNs $s \in S$ where $u \in s$ but $v \notin s$. Then the bit for $s$ in $r(u)$ is set to 1 but the bit in $r(v)$ is set to 0; hence, the bitsets are different. However, vertices are only clustered together if they have the same bitset value so there is a contradiction.

**Lemma 2.** If $v_1, \ldots, v_k$ is a path in the condensation graph $C$, then for any sub-ACN $s \in S$ : $v_k \in s \implies v_j \in s$.

**Proof.** Let $u$ be a minimal element in $C$ such that there is a path $v_k \sim u$ (without loss of generality, assume that a path can consist of a single vertex if $v_k = u$). There must be at least one unique $u \in C$ because $C$ is a DAG. The DECOMPOSE-MODULES algorithm builds a sub-ACN from $u$ by putting all vertices that are connected to $u$ in the sub-ACN. Since $v_k$ is connected to $u$, it is in the sub-ACN; since $v_j$ is connected to $v_k$ and $v_k$ is connected to $u$, $v_j$ is also in the sub-ACN.

**Lemma 3.** Given any two paths $p_\alpha, p_\beta \in P$ with $\alpha \geq \beta$, for any sub-ACN $s \in S$ and vertex $v_j^{(\alpha)} \in p_\alpha$, if $v_j^{(\alpha)} \in s$ then $\forall i \in S : v_i^{(\beta)} \in s 

We assume that $P = \{p_i\}_{i=1}^\pi$ is a sequence of $\pi \geq 1$ disjoint, simple paths from $C$. For notational purposes, we consider each of these paths $p_i$ to consist of the vertices $v_1^{(i)}, v_2^{(i)}, \ldots, v_k^{(i)}$. Below we show by contradiction that these paths cannot form a cycle. In order for a cycle to be form, the last vertex of each path $p_i$ must be clustered with the first vertex of the next path $p_{i+1}$ (i.e., $\forall i \in \mathbb{Z}_\pi, s \in S : v_k^{(i)} \in s \iff v_1^{(i+1)} \in s$).

**Lemma 3.** Given any two paths $p_\alpha, p_\beta \in P$ with $\alpha \geq \beta$, for any sub-ACN $s \in S$ and vertex $v_j^{(\alpha)} \in p_\alpha$, if $v_j^{(\alpha)} \in s$ then $\forall i \in S : v_i^{(\beta)} \in s$. This lemma is essentially a generalization of Lemma 2 to multiple paths.
Proof. Intuitively, this Lemma says that if a path contains a vertex \( v \) that is in a sub-ACN \( s \), then all vertices in all paths earlier in the sequence are also in sub-ACN \( s \). If \( v_j^{(\alpha)} \in s \) then, by Lemma 2, the first vertex \( v_1^{(\alpha)} \) of \( p_\alpha \) is also in \( s \). We are given that the last vertex of each path \( p_{\alpha-1} \) is clustered with the first vertex of the next path \( p_\alpha \). It is simple to see that we can continue applying Lemma 2 until we reach path \( p_\beta \) (since \( \alpha \geq \beta \)) and, by Lemma 2, all vertices \( v_i^{(\beta)} \in p_\beta \) can be shown to be in \( s \) also. (Since \( P \) is assumed to form a cycle, the restriction that \( \alpha > \beta \) is not actually required because the last vertex of the last path is clustered with the first vertex of the first path.)

**Theorem 1.** The DR hierarchy graph \( H \) is acyclic.

Proof. In order for \( P \) to form a cycle, at least one vertex in one of the paths must not be clustered with the other vertices (otherwise \( P \) forms a single vertex in the result graph \( H \) and is not a cycle). Let \( u \in p_i \) be this vertex that is not clustered with the other vertices; specifically, we assume by contradiction that \( u \) is not clustered with \( v_1^{(\pi)} \). Then by Lemma 1, there must exist a \( s \in S \) such that \( u \in s \) or \( v_1^{(\pi)} \in s \), but not both. We consider the two scenarios separately:

- \( u \notin s \land v_1^{(\pi)} \in s \)

  Lemma 3 shows that this scenario never occurs. Since \( u \in s \), \( u \in p_i \), and \( p_\pi \) is not before \( p_i \) in the sequence \( P \); Lemma 3 states that \( u \in s \) as well.

- \( u \in s \land v_1^{(\pi)} \notin s \)

  Since \( u \in p_i \) and \( p_1 \) is not after \( p_i \) in the sequence \( P \), Lemma 3 states that the first vertex \( v_1^{(1)} \) of \( p_1 \) is also \( s \). However, this contradicts the original assumption that \( u \notin s \) and \( v_1^{(1)} \) are clustered together, because we have \( v_1^{(\pi)} \notin s \) but \( v_1^{(1)} \in s \). By Lemma 1, this is a contradiction with the original assumption that \( P \) forms a cycle.

Therefore, \( P \) cannot be clustered to form a cycle and the DR hierarchy graph \( H \) is a DAG. Since \( H \) is a DAG, we guarantee that the corresponding DSM will can clustered into block triangular form and therefore meets the definition of DR hierarchy that we described earlier.

**Complexity Analysis**

To show the running time for our algorithm we first bound the size of its inputs. All \( |V[C]|, |S|, \) and \( |V[H]| \) are bounded by the number of variables in the ACN \( |V| \) because each vertex or sub-ACN must contain at least one variable. From this, we know that each of the first two for-loops of our algorithm will run in \( \Theta(|V|) \) times and the last for-loop runs in \( \Theta(|V|^2) \) time. Breadth-first search runs in linear time so the total running time of our algorithm is \( \Theta(|V|^2) \).
5.3 Evaluation

We implemented the DRH clustering algorithm as a plugin, called dr-hier, to our Parcae tool. To evaluate whether the DRH algorithm can correctly identify design rules, reveal their impact scope, and reveal independent modules, we use both Parnas’s small but canonical keyword in context (KWIC) and our Parcae tool as experimental subjects. We choose these two systems because we are familiar with their design and can easily tell whether the DRH is correct.

To evaluate the effectiveness of DRH in terms of identifying communication requirements and independent tasks, we use Apache Ant and Eclipse JDT as subjects. We explore both their code repositories and coordination media to identify tasks and analyze communications among members of the development team. We use this socio-technical information to establish evidence of the need for technical communication between developers that have worked on modules in different layers (as opposed to within the same layer) of the DR hierarchy.

5.3.1 Keyword in Context

To evaluate the correctness of our DRH clustering algorithm, we compare DRH-clustered DSMs with previously validated DSMs, in which the design rules are manually identified and the modules are manually clustered. We check whether the DRH-clustered DSMs identify the same set of DRs and independent modules as the previously published DSMs. If not, we investigate the differences.

We choose to compare against the DSM models constructed by Sullivan et al. [46, 50, 184] that model Parnas’s canonical and extensively-studied keyword in context (KWIC) [156] system. Cai and Sullivan [49, 50] presented a manually constructed ACN model of the system, which we use as input to our algorithm. In the DSMs presented by the previous work, all the environment variables were aggregated into a single module. To ease the comparison, we slightly modify our automatically-generated hierarchy by moving all the environment variables to a standalone module. Figure 5.6 shows a DSM of KWIC with this modified hierarchy. Since environmental conditions do not depend on design decisions, the DSM is still in block triangular form.

In comparing our DRH-clustered DSM with the manually-clustered DSM published in previous work [46, 50, 184], we notice only one difference: our DRH algorithm does not identify master ADT as a design rule, whereas the previously published DSMs did. We observe that the only dependent of master ADT (in the specified design of KWIC) is master_impl. Therefore, the classification of master ADT by our algorithm is correct and consistent with the definition of design rules: design
rules are meant to decouple subordinate modules but master ADT does not decouple two modules. As a result, our approach correctly classifies it as not being a design rule. we conclude that our approach accurately identifies the design rules and independent modules in the KWIC design.

5.3.2 Parcae

PARCAE is our ACN/DSM modeling and modularity analysis tool. It has about 10 KLOC of Java code, and employs a plugin architecture. To generate the DRH for PARCAE, we first reverse engineer the code base into a UML class diagram, then formalized the UML class diagram into an ACN. PARCAE then takes the ACN as input and computes the DRH-clustered DSM for itself.

The resulting hierarchy shows 5 layers. Of all the 149 variables, 33 of them (22%) are aggregated into 24 modules within the first layer. 94 variables (63%) are clustered into 51 modules in the last layer. After carefully examining each layer, we confirm that the design rules are correctly identified and the locations of these DRs reflect their importance level. For example, the first layer aggregates all of the most influential decisions, such as util Graph interface and nona Plugin interface.

We also find that the 51 modules in the last layer include 16 out of the total 18 plugins, each modeled as a block with at least two variables. For example, the DRH clustering plugin is modeled as 4 variables that are automatically aggregated into a module in the last layer. The fact that two plugins, Decomposer and cluster FileWriter are not in the last layer drew our attention. After
examining the dependencies shown on the DSM, we realized that unexpected dependencies had been introduced during evolution and maintenance activities of the Parcae software. Their effect is to push these two plugins up the hierarchy. Based on these observations from the Parcae DRH, we refactored those parts to improve its design.

In conclusion, as the designers and developers of Parcae, we confirm that the DRH clearly lays out the plugin architecture, and has even helped us identify some hidden poor design decisions.

5.3.3 Apache Ant

To evaluate the feasibility of applying our approach to a large-size, real-world project, we experimented with the popular, open-source Apache Ant project. Specifically, we selected release 1.6.5 as our target, and extracted its ACN model by reverse engineering its UML class diagram from the code base. The Apache Ant UML model contains around 500 classes and interfaces (including inner classes), and almost 2,500 inter-element relations.

We formalized an ACN from the UML class diagram, and derived a DRH from that ACN, in about 15 minutes on a 2.16 GHz Intel MacBook Pro laptop with 3 GB of RAM, and produced a DRH-clustered DSM that includes 12,596 pairs of dependencies among 1,200 variables.

DRH and Modularity Quality

According to Baldwin and Clark’s option theory [22], since no modules depend on independent modules in the last layer, they provide the option to be substituted with better implementations at minimal cost. Therefore, we can use the number of independent modules in a system as an indicator of its design quality. Despite having over 500 classes and 1000 variables in its DSM, the Apache Ant DRH consists of only 11 layers. When we compare this with, for instance, the maze game example, we see that although Apache Ant has 40 times the number of DSM variables, it has about only twice the number of layers in its DR hierarchy. This means that most modules in the system are aggregated horizontally within layers, and the corresponding development tasks can be highly parallelized. In addition, 52% of the tasks identified are in the bottom layer of independent modules, indicating that much of the system constitute options that can be freely substituted. Both of these DRH-based characteristics indicate that Apache Ant is well modularized, and easy to maintain and evolve. Our recent work [173] further investigates the use of the DRH for measuring software architecture modularity and stability.
Identifying Design Rules

When maintaining a software system, especially an unfamiliar one, it is crucial not to accidentally change parts of the system that are highly influential. We are interested to see whether a DRH can provide an ordering for design rules, in terms of their influence, so that developers can be aware of these most influential parts.

We examine the Apache Ant DSM and count the number of dependencies to determine if the identified design rules are indeed the most influential. The more other variables depend on a given design rule, the more influential it is. Even though a DSM is in block triangular form, it does not mean that the variables furthest to the left are the most influential. There may be some variables further to the right that depend on the left-most variables, whereas a larger portion of the rest of the system depends on them.

To verify if the identified design rules are the most influential, we use the DSM to count the number of dependencies upon each variable. Figure 5.7 shows a graph of the average and maximum number of dependencies for each layer of the DR hierarchy. We note that the vertical axis is in a logarithmic scale. From the graph, we see that the variables in layer zero have the most dependencies upon them. These high-level dependencies indicate that if a variable in layer zero were to change, it would impact much of the system. In other words, the variables in layer zero are highly influential on the rest of the system. Since both the maximum and average number of dependencies decrease at each layer of the hierarchy (except for between layer 2 and 3, where there is a slight increase in the average number of dependencies), we conclude that, almost always, each layer of the DR hierarchy for Apache Ant is less influential than the one before it.

Identify Concurrent Tasks

We set out to test whether the DR hierarchy can effectively identify modules that correspond to independent tasks for parallel work. Modules located within the same DRH level are supposed to be mutually independent, and thus constitute candidates for independent task assignments. To validate that assumption, we have one release of Apache Ant, and have examined the work and communication dynamics of its development team. Our study regards the development effort during the transition from Ant release 1.6.5. to the following release, that is, 1.7.0.

The method of our analysis descends from recent results on socio-technical congruence, which indicate how to establish and analyze the coordination requirements between developers working
on concurrent tasks [54]. A coordination requirement occurs when two developers, say, Alice and Bob, are assigned to concurrent development tasks, and those tasks require them to work on sets of software artifacts that present some dependencies. In those cases, some form of coordination between Alice and Bob is often necessary; for example, Alice and Bob may exchange some technical communication, which is often archived—particularly in open-source projects—and hence traceable. On the other hand, if Alice and Bob work concurrently only on mutually independent modules, their need for coordination can be greatly attenuated.

Based on the concept of coordination requirements, and the semantics assigned to DSM dependencies and DRH levels, we formulate the following hypotheses:

H1: *Technical communication between any two developers who are engaged in concurrent work on modules that are located within the same hierarchy level should be least intense.* This hypothesis aims at verifying that those modules are good examples of independent task assignments, and do not present involved Ant developers with coordination requirements.

H2: *Technical communication is likely to occur significantly more frequently “across layers” (i.e., between any two developers who are engaged in concurrent work on modules located in different layers of the hierarchy and are dependent on one another).* This hypothesis aims at verifying that the layered dependency structure provided by the DR hierarchy provides a good indication of where coordination is needed, and how it flows in the project.

For our analysis of Apache Ant, we collected all commits of Java code in the SVN repository during the development of release 1.7.0 (which lasted 19 months), as well as all message exchanges
within the developers’ mailing list by the 14 active committers for Apache Ant in the same period. We traversed the tree structure of each mailing list thread, and considered as a communication exchange between two developers any direct reply by one of them to a message posted by the other.

To identify development tasks, we had to devise a strategy to overcome the fact that the data set we could mine has only sparse information linking tasks to specific code commits within the source code repository. For example, only a small portion (about 15%) of the change requests, bug fixes, and other work items listed in the project’s Bugzilla are referenced within the metadata of Java code commits for Ant. Hence, we used a sliding time window to approximate the concept of parallel tasks with that of concurrent work. For each of the 866 commits involving any of the 1133 Java files, we computed a two week time window, and considered any commits by other developers within that window as concurrent work. (The choice of two weeks was suggested by examining the typical length of development tasks in the project, mediated by the frequency of communication exchanges observed within the developers mailing list.)

With this mechanism, all distinct pairs of developers committing Java code within the same time window would be considered as engaging in concurrent work. We were able to identify 742 such pairs. We then proceeded to eliminate those pairs in which either developer was responsible for commits that had to do with simultaneous blanket changes on a large number of files (more than 5% of the overall Java code base). Commits of such kinds are typically extraneous to actual development tasks. For example, they occur when versioning, copyright or licensing information needs to be updated in the comment section of all source files; or other trivial housekeeping activities on the code base.

That filter provided us with 653 developer pairs, upon which we carried out further analysis. First of all, we matched the commits by each developer to the variables represented in the DSM. That way, we came up with a list of the variables in each DSM module that could be affected by the changes made to the Ant code base by a pair of developers for a given time window. That provided us with a basis to locate concurrent work by a developer pair (say, Alice and Bob) within the DSM; we then proceeded to extract the following subgroups from the population of 653 pairs:

1. If Alice and Bob have done concurrent work affecting any pair of variables that have a dependency relationship and are located in different DRH layers, we count those pairs of variables and place them in the Across Layers (AL) category;
2. If Alice and Bob have done concurrent work affecting any pair of variables that are located in the same DRH module, we count those pairs of variables and place them in the *Same Layer Same Module* (SLSM) category;

3. If Alice and Bob have done concurrent work affecting any pair of variables that are located in different modules within the same layer of the DR hierarchy (which by definition have no dependency), we count those pairs of variables and place them in the *Same Layer Different Module* (SLDM) category;

To complement that information, we counted the number of mailing list exchanges between Alice and Bob within the same time window.

For any time window and for any pair of developers, it is of course possible that those developers have carried out work that falls into more than one of the above categories. That is important, because whenever a pair of developers has a count $> 0$ for either the SLSM or AL category, that indicates the presence of at least one coordination requirement. In fact, 347 out of 653 pairs have a count of AL $> 0$. There are also 144 pairs who have coordination requirements originating from SLSM work, but the vast majority of them are also included in the set with AL $> 0$: only 9 pairs have a count of SLSM $> 0$ and AL = 0. All in all, therefore, 356 pairs exhibited some form of coordination requirement. (We refer to them as the *CR group*.)

Similarly, we identified the 266 pairs who *exclusively* did SLDM work in some given time window, that is, whose SLDM count was $> 0$, and at the same time had both an SLSM and AL count of 0. (We refer to them as the *SLDM group*.)

Coming to technical communication data, out of the 266 pairs in the SLDM group, we found that 89 pairs exchanged mailing list messages, that is, about 33%. In contrast, about 53% of the pairs in the CR group exchanged messages. That percentage amounts instead to 43% when considering the overall set of 653 pairs.

First of all, in order to statistically confirm whether the observations above support our hypotheses H1 and H2, we set out to verify whether the difference in proportion among these groups could be considered significant. To that end, we carried out pairwise chi-square tests of proportion between the various groups, and between each group and the overall population. The results, which are summarized in Table 5.1, show how the differences in proportion are indeed statistically significant. In particular, it is confirmed with a very high degree of confidence that the proportion of communicating pairs in the SLDM group is significantly less than that of the CR group. Also the
proportion of communicating pairs in the SLDM group is significantly less than that of the overall population; similarly, the proportion of communicating pairs in the CR group is significantly more than that of the overall population. All of these observations are consistent with our hypotheses.

Table 5.1: Proportion Tests

<table>
<thead>
<tr>
<th></th>
<th>$\chi^2$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLDM &lt; population</td>
<td>7.709</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>SLDM &lt; CR</td>
<td>22.3014</td>
<td>&lt; $10^{-5}$</td>
</tr>
<tr>
<td>CR &gt; population</td>
<td>7.4064</td>
<td>&lt; 0.01</td>
</tr>
</tbody>
</table>

Since, however, a ratio of technical communication of 33% within the SLDM group seems high in absolute terms, given the absence of coordination requirements in that group, we carried out further statistical analysis, to try to understand whether we could consider such as 33% ratio as a sort of natural level of “chatter” within the communication channel provided by the developers mailing list, whereas higher ratios, such as 53% within the CR group could be indeed described as a consequence of doing interdependent concurrent work. To investigate that, we set out to verify the additional hypotheses below:

H3: The probability of communication between pairs in the SLDM group is not correlated to the amount of SLDM work they have concurrently carried out (i.e., the amount of different pairs of DSM variables affected by their commits in the time window considered). In contrast, the probability of communication between pairs in the CR group is correlated to the amount of dependent work they have concurrently carried out.

H4: The amount of communication (i.e., the count of messages, exchanged in the time window considered) between those pairs in the SLDM group that communicated is not correlated to the amount of SLDM work they have concurrently carried out. In contrast, the amount of communication exchanged between pairs in the CR group that communicated is correlated to the amount of dependent work they have concurrently carried out.

For the additional hypothesis H3 above, a point-biserial correlation test between the count of pairs of DSM variables touched by the 266 pairs in the SLDM group, and a boolean variable indicating whether those developer pairs communicated at least once provided us with an $r$-score of 0.00206, which is consistent with no correlation. That sharply contrasts with the result of the same statistical test performed on the 356 pairs in the CR group. The $r$-score is 0.299, which denotes a strong positive correlation; moreover, that correlation is extremely statistically significant, with $p < 10^{-8}$. These results confirm hypothesis H3, since they strongly suggest that the amount of SLDM work of a pair
of developers and their probability of communication do not influence each other; on the contrary, the amount of dependent work that produces coordination requirements for a pair of developers and the probability of technical communication between them are strongly linked.

For our additional hypothesis H4 above, we performed a Pearson correlation test between the count of pairs of DSM variables touched by the 89 pairs of developers in the SLDM group who exchanged messages, and the number of messages they exchanged. The $r$-score in this case is $-0.05626$, which is again consistent with no correlation. In the case of the 188 pairs of developers in the CR group who exchanged messages, the same statistical test returns an $r$-score of 0.189, which denotes a weak positive correlation. That correlation is quite significant statistically, with $p < 0.01$ ($p = 0.0048$). These result confirm hypothesis H4, since they suggest that the amount of SLDM work and the amount of communication do not influence each other; on the contrary, the amount of dependent work and the amount of communication are linked to each other.

From the comparative analysis above, we are confident to conclude that the 33% ratio of communication within the SLDM group is unlikely to be a consequence of any hidden coordination requirements among modules in the same layer of the DRH, or an artifact of some conceptual or technical error in the DRH construction process. The statistical evidence points instead to that ratio as being independent of coordination requirements, and we can probably regard it as a property of the communication channel we considered. The result also suggests that the dependency structure derived from the ACN model and clustered using DRH sufficiently approximates the corresponding coordination structure.

5.3.4 Eclipse JDT

To further corroborate the results from our study on Apache Ant, we performed another empirical study on the open source Eclipse JDT project. JDT is a long-running project that has consistently attracted a significant number of developers and code contributions, since it deals with a central element of the Eclipse IDE software. Our study explored the development period leading from release 3.2 to release 3.3, which lasted approximately one year. We selected this particular release in part for expediency, because the data sets of interest were used for the Mining Challenge at the Workshop on Mining Software Repositories (MSR) 2007$^4$ and are publicly available for scrutiny.

Our intention with the Eclipse JDT study was twofold. First, we wanted to confirm our findings from the Apache Ant case study in a different development context. Eclipse JDT represents a larger-

\footnote{http://msr.uwaterloo.ca/msr2007/challenge/}
Table 5.2: Comparison of Apache Ant and Eclipse JDT Project Measures

<table>
<thead>
<tr>
<th>Measure</th>
<th>Ant</th>
<th>JDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>1.6.5</td>
<td>3.2</td>
</tr>
<tr>
<td>Duration (months)</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>Code Files</td>
<td>621</td>
<td>4247</td>
</tr>
<tr>
<td>DSM Variables</td>
<td>1,200</td>
<td>10,903</td>
</tr>
<tr>
<td>DSM Modules</td>
<td>837</td>
<td>8,171</td>
</tr>
<tr>
<td>Dependencies</td>
<td>12,596</td>
<td>127,730</td>
</tr>
<tr>
<td>DRH Levels</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>Committers</td>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>Single File Commits</td>
<td>866</td>
<td>10,034</td>
</tr>
<tr>
<td>Concurrent Work Pairs</td>
<td>653</td>
<td>92,645</td>
</tr>
</tbody>
</table>

scale case study—in particular, with respect to the level of development activity and the size of the code base. This scale difference, in turn, influences the data set with respect to the number of DSM modules, the amount of potential concurrent work going on, etc. Table 5.2 presents a comparison of Ant and Eclipse JDT on these basic characteristics. Second, we want to verify that the assumptions we made in the Ant case study had no discernible effect on the validity of the results.

Perhaps the most significant assumption we made in analyzing Apache Ant involved our need to approximate concurrent development tasks by means of a sliding time window. This assumption was made because of the lack of consistently reliable information linking descriptions of project tasks to records of development. In the case of Eclipse JDT, however, this issue can be overcome due to the conventions observed by the Eclipse open source community in using the available project management support technology. In Eclipse JDT, code commits logged in the CVS repository are routinely linked to work items listed in the Bugzilla change request repository, since developers conventionally include the change request ID in the comment message for their commits. For the release we considered, 66% of the Java code commits contain such a reference; and out of a total of 2,680 change requests which were both created and closed (with different states) within the development period we considered, 1099 (or 41%) are referenced in one or more code commits. These conventional links enable us to precisely associate commits to software artifacts in the corresponding development tasks. We used those 1099 change requests as our development tasks, and define any two tasks whose duration overlapped as concurrent work items. This eliminated the need to compute and use a sliding temporal window to approximate concurrent work, as we did for Apache Ant.

When we examined the temporal information about these 1099 development tasks, we found that the task duration was on average about 46 days, which spans a considerable portion of the overall development period considered (approximately 13%). Looking at the distribution of durations,
which, as expected, is highly skewed, there are relatively few very long tasks (several months of duration), and the majority of tasks are significantly shorter than the average. We were concerned that those few very long tasks could disproportionately influence our analysis, because they would be considered as concurrent with respect to a large number of other tasks, even though they likely remained inactive for a large portion of their duration. To address this issue, we identified a subset of development tasks with more uniformity in their duration, by trimming off the 10% longest-duration tasks. This reduced set included 989 tasks, with an average duration of less than 27 days (covering approximately 7% of the considered development period). The rest of our analysis—as described below, except when noted otherwise—was first performed on this reduced set. We then repeated the analysis with the full 1099 task set, to understand if our decisions had an effect on the findings.

We analyzed the technical communication channels made available to developers—collecting messages from both the developers’ mailing list and the change requests. Since the Eclipse JDT developers rarely used the mailing list for communication exchange, we only identified 17 pairs of coordinations from the mailing list. On the other hand, developers actively communicated using Bugzilla’s infrastructure for commenting on change requests. As these comments were targeted at specific change requests, their content were more likely to pertain to the task. Hence, we considered these comments as higher quality technical communications, compared to the messages from the mailing list.

Since comments on a change request are not explicitly threaded, it was difficult to attribute one-to-one communication exchanges from those comments. We followed a common convention from similar cases; that is, we considered all developers who comments on a change request to have pairwise communication exchanges (i.e., forming a clique of communication exchanges among the developers). This represented a very different choice with respect to how we constructed the communication network for Apache Ant, and thus served the purpose of testing the effect of that kind of assumptions on the validity of results.

Next, we mined the Eclipse JDT CVS repository: during the development of release 3.3, there were 12,693 single file commits. As per standard practice, we again eliminated commits that had to do with simultaneous blanket changes involving a large number of files (more than 5% of the overall Java code base). That left us with 10,034 commits. Finally, all distinct pairs of developers committing some changes to Java code within the same two overlapping tasks were considered as engaging in concurrent work. We identified 96,245 such pairs. We also considered all distinct pairs of developers committing Java code within the same task, but there were only 83 occurrence of such
pairs in the set of 989 task we considered—meaning that tasks in JDT are normally assigned to and resolved by an individual developer.

After extracting all the data described above from the Eclipse JDT repositories, we broke down the data set to obtain the CR and SLDM groups from all the 96,245 developers pairs involved in concurrent work, in the same way we did for the Apache Ant case study. The CR group for Eclipse JDT includes 8,259 developers pairs, whereas the SLDM group includes 34,779 pairs. We then set out to verify the same hypotheses (H1 through H4), enunciated for the Apache Ant case study, on this new data set. We repeat the four hypotheses here, for convenience:

H1: Technical communication between any two developers who are engaged in concurrent work modules that are located within the same hierarchy level should be least intense.

H2: Technical communication is likely to occur significantly more frequently “across layers” (i.e., between any two developers who are engaged in concurrent work on modules located in different layers of the hierarchy and are dependent on one another).

H3: The probability of communication between pairs in the SLDM group is not correlated to the amount of SLDM work they have concurrently carried out (i.e., the amount of different pairs of DSM variables affected by their commits in the time window considered). In contrast, the probability of communication between pairs in the CR group is correlated to the amount of dependent work they have concurrently carried out.

H4: The amount of communication (i.e., the count of messages, exchanged in the time window considered) between those pairs in the SLDM group that communicated is not correlated to the amount of SLDM work they have concurrently carried out. In contrast, the amount of communication exchanged between pairs in the CR group that communicated is correlated to the amount of dependent work they have concurrently carried out.

With respect to hypotheses H1 and H2, we tested the statistical significance of the difference in proportion of communicating pairs across the SLDM group, the CR group and the whole population. The results strongly confirm those hypotheses since the differences are statistically very significant. Out of 8,529 CR pairs 6,041, or 70.8%, communicate while only 9,887 of 34,779, or 28.4%, SLDM pairs communicate. When looking across the whole pairs population, 32,846 of 96,245, or 34.1%, communicate. So, once again SLDM pairs communicate less often than the overall population while CR pairs communication significantly more often. Table 5.3 recaps the outcome of these tests.
Table 5.3: JDT Coordination Proportion Tests

<table>
<thead>
<tr>
<th></th>
<th>$\chi^2$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLDM &lt; population</td>
<td>377.336</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
</tr>
<tr>
<td>SLDM &lt; CR</td>
<td>5294.055</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
</tr>
<tr>
<td>CR &gt; population</td>
<td>4520.027</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
</tr>
</tbody>
</table>

For hypothesis H3, we first looked at the correlation between the count of DSM variables touched by the 34,779 developer pairs in the SLDM group, and a boolean variable indicating whether those developer pairs communicated at least once. Since the distribution of DSM variable counts strays significantly away from normality, we log-transformed that data, and carried out a point-biserial correlation [97], providing us with an $r$-score of 0.01181264 and a $p$-value of 0.0138—which is consistent with no correlation. To further confirm H3, we carried out the same statistical test on the 8,529 developers pairs in the CR group. The resulting $r$-score is 0.3913705 and the $p$-value is $< 2.2 \times 10^{-16}$, which denotes a strong positive and extremely significant correlation.

Regarding hypothesis H4, we considered the correlation between the count of DSM variables touched by the developers pairs of in the SLDM group who exchanged messages, and the number of messages they exchanged. Since both sets of data are not normally distributed, we computed their Spearman correlation [177]. The $r$-score in this case is 0.02523474, with a $p$-value of $< 10^{-6}$, which again is consistent with no correlation. In the case of the 8,529 pairs of developers in the CR group who exchanged messages, the same statistical test returns an $r$-score of 0.3993827, with $p < 2.2 \times 10^{-16}$, which again denotes a strong and highly significant positive correlation.

To further fortify our confidence in the results above, and ensure that we did not skew the results by removing the 10% longest-duration tasks, we repeated the same statistical analysis on the whole data set. With respect to hypotheses H1 and H2, we observed in this larger data set the following:

- 17,763 CR pairs, of which 11,715, or 66%, communicate;
- 80,243 SLDM pairs, of which 23,011, or 28.6%, communicate;
- an overall population of 200,069 developer pairs engaging in concurrent work, of which 68,501, or 34.2%, communicate.

Statistical results related to these proportion are reported in table 5.4.

With respect to hypothesis H3, the correlation between SLDM work and the probability of communicating has an $r$-score of 0.04590161 and a $p$-value $< 2.2 \times 10^{-16}$, while the correlation between number of CRs and probability of communicating has an $r$-score of 0.3540559 with $< 5 \times 10^{-6}$, which is the minimum $p$ value that R, our statistical package of choice, can compute.

\[5\]
Table 5.4: JDT Coordination Proportion Tests (all tasks)

<table>
<thead>
<tr>
<th></th>
<th>$\chi^2$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLDM &lt; population</td>
<td>805.5286</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
</tr>
<tr>
<td>SLDM &lt; CR</td>
<td>8831.03</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
</tr>
<tr>
<td>CR &gt; population</td>
<td>7051.457</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
</tr>
</tbody>
</table>

$2.2 \times 10^{-16}$. With respect to hypothesis H4, the correlation between SLDM work and amount of communication has an $r$-score of 0.05911598 and $p < 2.2 \times 10^{-16}$, while the correlation between number of CRs and amount of communication has an $r$-score of 0.3721005 and $p < 2.2 \times 10^{-16}$.

All of these results are clearly similar and fully in line with those obtained from the data set without the longest 10% of tasks.

**Outcome of the Eclipse JDT case study.** As a conclusion, from the Eclipse JDT data set we obtained confirmatory statistical evidence of the findings observed in the Apache Ant project, and a further validation of our hypotheses regarding the value of the DRH structure for identifying independent—and hence parallelizable—development tasks. This validation is particularly significant, since these two case studies are diverse in scale, and cover different set of assumptions that are likely to impact considerably the outlook of this kind of analysis, such as the modeling of technical communication exchanges, and the definition of concurrent work.

**Investigating “In Excess” Communication**

Given the scale of the Eclipse JDT case study, and the volume of data it yielded, we decided to explore also another related issue—that is, the characteristics of those pairs of developers that have no CR, but entertain some amount of technical communications. In the JDT data set, we found 26,805 pairs of people, out of a total 96,245, who exchange some communications without a CR. Of those, 14,448 pairs participated in technical communication exchanges on the development of Eclipse JDT v.3.3 without contributing any code. However, the remaining 12,357 pairs committed code without incurring in any CRs, but still carried out technical communication.

Since the comments are posted to specific change requests, we can largely discount that those communications are completely extraneous to the development task at hand, or to other tasks that are somehow related. However, in these cases, we cannot find a direct explanation for these exchanges within the DRH and its inter-module dependencies. It is therefore an interesting research question to isolate those cases in which technical communication appears to be “in excess” of what we can account for with the DRH.
There are many possible technical reasons why these communications may occur (e.g. they may capture bug triage discussions, task assignment negotiations, managerial decisions on whether to include the code developed for of a task in an upcoming release). Those reasons may not be directly traceable to the technical content of the work being done and the code developed for that task. However, even those kinds of technical communication can be construed as forms of coordination between members of the development team who may have some stake in different areas of the project. Therefore, we are interested to see whether they occur more frequently between pairs of developers that tend to work on interdependent parts of the modular structure. We therefore formulated the following hypothesis:

**H5:** Technical communication in absence of CRs occurs more frequently between developers who carry out their respective work across the DRH layers, with respect to developers who carry out their development work within the same DRH layer. This hypothesis postulates that the hierarchical layout of the DRH structure is indicative of coordination needs, even when those are “soft” (i.e., not explicitly reified in some design dependencies between specific modules) like in the examples we offered above.

To investigate H5, we subdivided the set of 12,357 developer pairs with “excess communication” according to the work relationship they have: we identified a subset of 2,470 pairs, which only carried out work across layers, that is, they are exclusively in an AL work relationship; similarly, we identified a subset of 129 pairs who are exclusively in an SLDM work relationship. We decided to instead discard the remaining 9,758 pairs who are in a mixed AL/SLDM work relationship. We display the difference between the AL-only and SLDM-only subsets as a box-plot in Figure 5.8.

To measure statistically the intuitive difference made evident by the box-plot, we used a Mann-Whitney-Wilcoxon test [138], given the non-normal distribution of communications in both populations. We tested whether the amount of communication observed in the AL-only population is greater than that observed in the SLDM-only population, and obtained a $W$-score of 187,224.5 with a $p$-score $< 10^{-3}$, which is statistically quite significant. This result validates hypothesis H5, and—as such—provides further evidence of how the hierarchy of levels approximate the coordination structure of an organization, even with respect to “soft” coordination needs. That is conceptually consistent with the design rules abstraction from which the DRH derives its semantics. Design rules, which appear at the top level in a DRH, represent the principal design decisions in a system, and influence all other design decisions. By extension, design decisions at any given level of the DRH
are potentially influenced by other design decisions in higher layers; therefore, when operating on modules located within a given level, a developer may need to reconcile her decisions with others who are working on those higher layers, sometimes even in cases where there are no explicit inter-module dependencies. In contrast, developers operating within the same DRH layer can take their design decisions more autonomously.

5.4 Discussion

The use of our DR hierarchy approach to identifying tasks for parallel development assumes the availability of an initial design or architectural prior to implementation. This requirement implies that our DRH approach is better suited for software development methodologies with an up-front design phase (e.g. waterfall, iterative). However, our approach can also offer benefits to some agile methodologies that do not have the full design available prior to implementation. Even organizations that use agile methodologies of development often create a high-level architectural design of the software system in order to enable the non-functional benefits that software architecture provides (e.g. ease of maintenance). Our DR hierarchy can be applied to these high-level designs to identify the major components that can be further designed and implemented in parallel. As each component is further designed, the DRH approach can be recursively applied to identify potential concurrent tasks. Additionally, the identification of design rules by the DRH highlights the parts of the system that should remain stable as the system evolves in order to minimize the impact of a change.
5.4.1 Threats to Validity

Our most important assumption is about the medium and method we used to analyze communication flows. That analysis is, first of all, necessarily partial (we can only take into account those communication acts whose traces are persisted in some archive, and hence observable). Also, we could have captured messages that are not related to any development work whatsoever. However, our focus on the developers’ mailing list and change request comments ensures that the large majority of discussions would be of a technical nature, and centered upon the development tasks at hand. That is corroborated, for example in the Ant case study, from our manual examination of random thread samples, and from the fact that we could only identify 8 mailing list exchanges between developer pairs that seemed to occur outside of some development activity (i.e., with either developer not engaged in any development work during the considered time window).

We mined mailing list threads (i.e., a many-to-many communication medium) which we had to map onto one-on-one exchanges. For that, we leveraged—as discussed earlier—the tree structure of the thread. That decision is of course a simplification of how communication flows in a mailing list, and represents a major factor in shaping our results. In the absence of reliable knowledge of the intent of the poster, and/or about the “read” activities of developers consulting mailing list threads, the only additional insight could be provided by a deep semantic understanding of the content of each message. However that kind of analysis is definitely challenging and very costly, and probably itself error-prone to an extent. It is nevertheless important to extend the analysis to other communication channels that may be available, to understand whether assumptions on how communication flows can skew results in some way.

Although we used modification requests as the source for communications in the Eclipse evaluation, we cannot guarantee that the MRs used in the evaluation are not biased. The studies by Bird et al. [19,29] show the MRs that have associated change sets may not be representative of all the MRs in the system. In other words, the subset of MRs that we looked at (i.e., the MRs with associated change sets so that we can identify the files involved in the task) may have different communication patterns than the MRs that we did not examine. However, by using two different approaches to identify tasks, and in particular their concurrency relationships, in our two case studies, we aimed to minimize the effects of such bias.
5.4.2 Future Work

It is generally accepted that designs often change before implementation is completed. Our DRH algorithm can quickly and easily be re-run to generated a new partitioning of tasks for the changed design; however, the new structure of tasks may conflict with tasks that are already assigned for development. For example, two modules under concurrent development may no longer be entirely independent (in the same DRH layer) as a design changes. To address this type of scenario, we plan, as future work, to extend the DRH clustering algorithm in order to allow addition constraints on the partitioning of elements (e.g. variables $A_{\text{impl}}$ and $B_{\text{impl}}$ must be in the same module). Automatically identifying such constraints based on an existing DRH and set of modules that are under implementation is also a future work.

Following the design stability measurement techniques of Sethi et al. [172,173], we can extend the DRH to provide quantitative measures on the independence of tasks (rather than just a boolean value of whether modules are independent). The basic idea is to include environmental decision (e.g. requirements) as additional variables in the ACN and link the environment variables to design decision variables via logical constraints in the ACN (following the work of Cai and Sullivan [49]). This inclusion of environmental decisions in the ACN allows us to identify the parts of the design that are likely to be affected by a change in requirements. Following the work of Sethi et al. [172,173], we can assign a volatility measure (i.e., probability the decision will change in the future) to each environment variable. Then, we can compute a volatility measure for each design rule. Since the stability of a design rule enables independent task assignments [22], the modules that depend on the same design rule for independent development are affected by the design rule’s volatility. If a design rule is highly volatile, it is likely to change and its dependent modules are unlikely to be independent tasks.

5.5 Conclusion

In this chapter, we contributed a design rule hierarchy (DRH) that can be used to predict coordination requirements, to suggest task assignments that maximize parallelism, and to reveal the impact scope of design rules. We evaluated this hierarchy using the Apache Ant, Eclipse JDT, Parcae, and KWIC systems as case studies. By investigating the source code repositories, developers’ mailing lists, and change request messaging archives, we showed that technical communication among developers working on different modules in the same hierarchy layer is significantly less in-
tense than that required by developers working across layers, supporting coordination predictions enabled by our DRH. Using PARCAE and KWIC, we also show that the hierarchy faithfully reveals design rules and their level of importance. These experiments therefore demonstrate the potential of our DRH model for reasoning and making predictions about the interplay between design structure, coordination structure and task assignment.
6. Generalizing Evolutionary Dependencies

Using software revision history (i.e., evolutionary coupling) for various kinds of software analyses has been shown to be effective. For example, empirical studies [53, 54, 69] suggest that evolutionary dependencies are more accurate estimates of coordination requirements than structural dependencies. In particular, software change impact analysis\(^1\) (IA) [30] exemplifies another analysis technique where evolutionary dependencies outperform structural dependencies. Traditionally, change impact analysis is often performed based on the software dependency structure. The problem is that semantically coupled components may not structurally depend on each other. Recently, researchers have leveraged revision histories to more effectively identify semantically coupled components (i.e., evolutionary coupling) by checking how components historically change together.

Although historical data has been shown to be a useful resource for various analyses, a question that arises is, *how long of a history is enough?* If taking a very long history into consideration, the software may have changed so that the coupling discovered in the distant past may not be valid anymore [80,158]. If the history is too short, the revision data may not be sufficient to reveal semantic coupling. The answer to this question also depends on the nature and maturity of the specific system. For example, the length of history needed to accurately predict change impact would be different for a relatively new system that is frequently refactored than for a long established system with a stable architecture. The problem is that existing history-based analysis techniques do not account for the temporal dimension or the evolution of software architectures.

To improve the accuracy of history-based analysis techniques, we contribute a formalized generalization of evolutionary dependencies based on probability theory, which we call *stochastic dependencies*. This formalization defines a family of dependency types. Each member of the family weights history data differently according to their recency and frequency. Our stochastic dependency family can be used to augment prevailing history-based analysis techniques by selecting the member of the family that fits the project.

Our stochastic dependency framework is based on the Markov chain model [58], a probabilistic model often used in artificial intelligence for determining the expected result of a random variable/event. Our assumption is that each transaction in revision history can be modeled as a random variable/event. Our assumption is that each transaction in revision history can be modeled as a random variable/event. Our assumption is that each transaction in revision history can be modeled as a random variable/event.

\(^1\)Change impact analysis (also called change propagation analysis and change scope analysis) is the method of determining what software elements are likely to change (called the change scope or the impact scope) when, or after, a certain set of elements (called the starting impact set) changes [30].
variable. To address the temporal issue of changing dependencies, our approach first limits the length of historical data used for analysis. This limited sequence of transactions create a sliding window that resembles a Markov chain (discrete $k$-th order Markov process) in which each state of the chain is a transaction from revision history.

From the Markov chain, we can compute the probability that two components are coupled (i.e., they will both be changed in the next transaction). This probability value can be used to reason about whether a component will be in the impact scope of a change. To account for the importance of recent history over distant history, our model for computing this dependency probability uses a smoothing function to control how much each transaction contributes to the predicted probability. The parameters to our framework (i.e., the length of history to use and the type of smoothing function) allow for the definition of a family of stochastic dependency types. To evaluate our approach, we conduct change impact analysis on 86 releases of five open source systems by varying the selection of parameter values of stochastic dependencies. Our results show that the stochastic-based change impact analysis outperforms two traditional structure-based IA approaches and an existing history-based IA approach in all five systems.

The rest of this chapter is organized as follows: Section 6.1 briefly surveys related work. Section 6.2 formally defines stochastic dependencies and how to use them for impact analysis. Section 6.3 presents our evaluation method and results. Section 6.4 discusses threats to validity and future work. Section 6.5 concludes.

6.1 Related Work

In this section, we review related work and differentiate our impact analysis technique from existing approaches.

6.1.1 Impact Analysis

Numerous IA techniques have been proposed. Below, we discuss several representative categories of IA approaches.

Structure-based Impact Analysis

Various IA techniques have been developed using software dependency structures. For example, Briand et al. [37] and Elish and Rine [81] have proposed IA techniques based on the metrics of
Chidamber and Kemerer [57]. Robillard [163] and Rajlich [161] presented algorithms based on dependency graph structures. Although Robillard’s algorithm is not specifically designed for impact analysis, it can be used for IA based on the weights assigned to each element. To capture impact that cannot be easily identified through syntactic analysis, some IA approaches (e.g. Arisholm et al. [14], Apiwattanapong et al. [13]) use dynamic (i.e., instance-based rather than class-based) analysis to measure coupling. While impact analysis techniques based on structural coupling can be effective, some semantic relationships between software elements can only be revealed through analysis of revision history.

**History-based Impact Analysis**

Similar to our approach, Ying et al. [202] and Zimmermann et al. [205] also leverage revision histories to perform IA based on association rules that identify evolutionary (or logical) coupling [91] between software elements. An association rule, of the form $a \Rightarrow b$, predicts that if $a$ is changed then $b$ will likely also be changed. As explained in Chapter 2, these rules are prioritized based on the heuristics of *support* and *confidence*. Support is defined as the number of transactions that $a$ and $b$ occur together in revision history. Confidence is defined as support, divided by the number of times that $a$ occurs (with or without $b$). With minimum support and confidence thresholds, association rules are selected to predict the impact scope of a change starting from $a$.

Evolutionary dependency approaches often consider the transactions in the overall revision history to be a multiset and disregard the temporal sequence (i.e., ordering) of the transactions. Two approaches that include temporal information in IA include the work of Bouktif et al. [31] and CecCcarelli et al. [55]. Different from our approach of using Markov chains, they use techniques to infer cause-effect relationships from the temporal data, rather than to filter out obsolete history data (i.e., similar to sequential pattern mining).

While these existing history-based techniques are effective at identifying semantic dependencies between software elements, refactorings that remove these dependencies may cause the approaches to overestimate the impact scope. Since the support heuristic never decreases, once support passes the minimum threshold, a dependency between two elements continues to exist. To address this issue, the confidence heuristic can be used in conjunction with support. However, as the number of transactions becomes large, confidence only minimally changes with each transaction. Our stochastic dependency framework addresses this issue by limiting the length of historical information analyzed and uses a smoothing function to emphasize more recent history over distant history.
Cataldo [51] explored the question of how many months of revision history are enough to accurately compute evolutionary dependencies and coordination requirements. He constructed task dependency matrices on monthly increments to find when a matrix no longer significantly differs from a previous matrix. Although he was able to find a fix point (19 months) when the dependencies stabilized, the results cannot be generalized to other software systems. Recent architectural refactorings can significantly alter the structure of dependencies, invalidating previous revision history. To address the issue of refactorings, our approach uses a smoothing function to emphasize more recent transactions over older transactions.

Probabilistic Impact Analysis

Recently, several impact analysis techniques have been proposed that are based on probability theory. For example, Tsantalis et al. [194] define probabilities of impact based on structural relations between software elements in object-oriented designs. Abdi et al. [1] use a Bayesian network (a probabilistic model) with structural coupling metrics to construct the inference model. Mirarab et al. [143] also use a Bayesian network and combine structural coupling with historical data in constructing the network. While our approach also uses a probabilistic model (Markov chain), stochastic dependencies use the temporal sequence of transactions in revision history to account for changes to the dependency structure over time.

6.1.2 Markov Processes

Markov processes (of which Markov chains are a specific type) are probabilistic models that are widely used in artificial intelligence (e.g. reinforcement learning [165]) and various other computing fields (e.g. web search engine algorithm [58]). Markov processes have also been applied to software engineering (e.g. generate test inputs [197], classify software behavior [34], predict component reliability [56]). To the best of our knowledge, Markov processes have not yet been applied to computing evolutionary dependencies.

6.2 Approach

In this section, we first present the definitions and mathematical notations to formally define the stochastic dependency. Then, we describe a method for computing the dependency value.
6.2.1 Definitions and Background

Let \( E = \{ e_i \}_{i=1}^{N} \) be the set of software elements of the system under consideration, where \( N \) is the total number of elements.\(^2\) Let \( T = \{ t_i \}_{i=1}^{M} \) be the sequence of revision history transactions, each involving a subset of software elements (i.e., \( \forall t_i \in T : t_i \subseteq E \) ), where \( M \) is the length of \( T \).

We can view \( T \) as a stochastic process, where each \( t_i \) is a discrete random variable with domain \( 2^E \). For any element \( e \in E \), let \( T(e) = \{ t_i(e) \in T | e \in t_i(e) \} \) be the subsequence of \( T \) involving \( e \).

Without loss of generality, we use separate indexing sequences for \( T \) and \( T(e) \)—in other words, \( t_1(e) \) is not necessarily the first element of \( T \), \( t_2(e) \) is not necessarily the second element of \( T \), etc. Let \( M(e) \) be the length of \( T(e) \).

Given two elements \( a, b \in E \), let \( C^{(a,b)} = \left\{ X_{i}^{(a,b)} \right\}_{i=1}^{M(a)} \) be a stochastic process that models whether \( b \) is in the transactions that involve \( a \):

\[
X_{i}^{(a,b)} = \begin{cases} 
1 & \text{if } b \in t_i^{(a)} \\
0 & \text{otherwise}
\end{cases}
\]

We define the stochastic dependency \((b, a)\) at time \( \tau \) as \( \Pr \left( X_{\tau}^{(a,b)} = 1 \right) \). The method for computing this probability varies among the different types of stochastic dependencies. Unlike some existing dependency definitions, in which a dependency either exists or does not, we define that an element is stochastically dependent upon another with a continuous probability of range \([0, 1]\). Given a starting impact set \( a \), we compute its stochastic dependents from all other elements \( b \in E \setminus \{a\} \) to determine if \( b \) will be in the impact scope of \( a \). In this chapter, we use the terms change scope and impact scope interchangeably.

With the probabilistic value of the stochastic dependencies, we can reason about which elements are expected to be in an impact scope. While we can simply sort the software elements by decreasing probability values and present the list to a maintainer, this continuous range also allows for various techniques to reduce the number of elements presented to the maintainer. For example, we can define a minimum probability threshold and consider all elements above that threshold to be in the impact scope. Alternatively, we can use randomized rounding \([160]\) to select the likely impacted elements. Next, we formally define a method for computing stochastic dependency values.

\(^2\)Although software elements may be added and deleted over time, we use \( E \) to refer to the set of elements in the software at the time of interest.
6.2.2 Computing Stochastic Dependencies

Given a sequence of \( \tau - 1 \) transactions in revision history involving an element \( a \), we have defined the probability that another element \( b \) will be involved in the next transaction involving \( a \) as the *stochastic dependency* from \( b \) to \( a \) at time \( \tau \). For each of the \( \tau - 1 \) transactions involving \( a \), let \( x_i \) be the value of \( X^{(a,b)}_{\tau-i} \) (i.e., \( x_i \) indicates whether \( b \) and \( a \) occurred together in the \((\tau-i)\)-th transaction involving \( a \)). Then we define the probability that \( b \) is stochastically depend on \( a \) when the \( \tau \)-th transaction involving \( a \) occurs as follows:

\[
\Pr \left( X^{(a,b)}_{\tau} = 1 \right) \equiv \Pr \left( X^{(a,b)}_{\tau} = 1 \mid X^{(a,b)}_{\tau-1} = x_1 \land \cdots \land X^{(a,b)}_1 = x_{\tau-1} \right)
\]

As a first step in accounting for the evolution of software, we consider only the latest \( k \) transactions involving \( a \) for analysis. We emphasize that these are the last \( k \) transactions that software element \( a \) is involved in, but not necessarily the latest \( k \) transactions in the revision history. Selecting an appropriate value for \( k \) can affect the accuracy of impact analysis. If \( k \) is too large then dependencies may have been removed during evolution. On the other hand, if \( k \) is too small then semantic dependencies between components may not be detected. We investigate the effects of various \( k \) values in our evaluation in Section 6.3.

Only considering the latest \( k \) transactions creates a sliding window that resembles a discrete \( k \)-th order Markov process (Markov chain), which leads to our method of computing the stochastic dependency probability. A Markov chain is a stochastic process with a property that the next state depends only on the current state and a finite number of previous states, but not the entire history of states. A \( k \)-th order Markov chain depends on the current state and the \( k-1 \) previous states—or formally, \( \Pr(Y_i \mid Y_{i-1}, \ldots, Y_1) \equiv \Pr(Y_i \mid Y_{i-1}, \ldots, Y_{i-k}) \).

A prevailing model [159] for high order Markov chains defines the probability for a next state to be based on a linear combination from the previous states. Based on this prevailing model, we derive a method to compute *stochastic dependencies* that is also based on a linear combination of previous states:

\[
\Pr \left( X^{(a,b)}_{\tau} = 1 \right) \equiv \sum_{i=1}^{k} \lambda_i X^{(a,b)}_{\tau-i}
\]

where \( \lambda_i \in [0,1] \)

and \( \sum_{i=1}^{k} \lambda_i = 1 \)
Based on this model, every time \( b \) changes with \( a \) in a transaction, we gain evidence that \( b \) depends on \( a \) and this evidence contributes to the computed probability. Intuitively, if \( b \) often changes with \( a \) recently, then it is more likely that it will change with \( a \) in the near future. On the contrary, if \( b \) only changes with \( a \) in the distant past, but not recently, it is more likely that this dependency has been removed during evolution. To capture this temporal phenomenon, we use a monotonically decreasing smoothing function,\(^3\) \( \lambda \), to account for software evolution and weigh more recent transactions more heavily than older transactions. Table 6.1 shows several functions that can be used as the smoothing function \( \lambda \). The first column shows a constant function in which all the transactions are weighed the same regardless of how long ago they occur, as with the traditional evolutionary dependency definition. The second column shows a linear function, indicating that the usefulness of a transaction in terms of determining stochastic dependency decreases linearly over time. The third column shows a sinusoidal function that weighs the most recent transactions similarly, and older transactions less and less. The last column shows an exponentially decaying function, which models a rapid decreasing of the usefulness of a transaction over time. These last three functions weigh past transactions less heavily than recent transactions.

By using a limited history \( k \) and a decreasing \( \lambda \) sequence, our stochastic dependencies potentially can recover more quickly from refactorings, which may significantly alter the dependency structure of a design, than traditional evolutionary dependencies that only use confidence to detect such changes.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Linear</th>
<th>Sinusoidal</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>( \frac{2(k - i + 1)}{k^2 + k} )</td>
<td>( \frac{\cos\left(\pi k^{-1}(i - 1)\right)}{k + 1} + 1 )</td>
<td>( 2^{-i} \times \frac{2}{k2^k + 1} )</td>
</tr>
</tbody>
</table>

Our stochastic dependency definition is a generalization of traditional evolutionary dependencies because they represent a specific \( k \) value and \( \lambda \) sequence for stochastic dependencies. For example, to use a minimum support value of \( \sigma \) and minimum confidence of \( \chi \) in determining traditionally-\( ^3 \)Technically, \( \lambda = \{\lambda_i\}_{i=1}^{k} \) is a sequence of \( k \) values but it can be intuitively understood as a function that maps to the appropriate sequence value. Hence, we interchangeably refer to it as either a sequence or a function.
defined evolutionary dependency, we can compute the evolutionary dependency \((b, a)\) at the time when \(\tau\)-th transaction involving \(a\) occurs using our stochastic dependency model by letting \(k = \tau\) (considering all the existing transactions involving \(a\)) and \(\lambda = \{1/k\}_{i=1}^k\) (treating these transactions equally). Then a evolutionary dependency \((b, a)\) is said to exist at time \(\tau\) if:

\[
\Pr \left( X_{\tau}^{(a,b)} = 1 \right) \geq \max \left\{ \chi, \frac{\sigma}{\tau} \right\}
\]

6.2.3 Example

We use a concrete, hypothetical example to demonstrate how to conduct change impact analysis based on stochastic dependencies. Suppose we have the following sequence of transactions in our revision history: \(\{a, b\}, \{a, c\}, \{d\}, \{a, b\}, \{a\}, \{a, b, c\}, \{b, d\}, \{a\}, \{a, d\}, \{c\}, \{a, c\}, \{a\}\). Now we want to perform IA for a anticipated change to \(c\). If using a linear \(\lambda\) sequence with \(k = 5\), we first find the last five transactions that involve \(c\): \(\{a, c\}, \{a, b, c\}, \{c\}, \{a, c\}\). Since \(c\) has only been involved in four transactions so far, we only use the four available transactions. Then we determine if any of the other files \(\{a, b, d\}\) are expected to be in the impact scope of changing \(c\) by applying the stochastic dependency formula:

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & \{a, c\} & \{c\} & \{a, b, c\} & \{a, c\} & \Pr \\
\hline
a & \lambda_1 \cdot 1 & \lambda_2 \cdot 0 & \lambda_3 \cdot 1 & \lambda_4 \cdot 1 & 0.6667 \\
\hline
b & \lambda_1 \cdot 0 & \lambda_2 \cdot 0 & \lambda_3 \cdot 1 & \lambda_4 \cdot 0 & 0.2 \\
\hline
d & \lambda_1 \cdot 0 & \lambda_2 \cdot 0 & \lambda_3 \cdot 0 & \lambda_4 \cdot 0 & 0 \\
\hline
\end{array}
\]

If we use a threshold-based rounding scheme with the minimum threshold to be 0.5, then we would predict that only \(a\) is likely to be in the impact scope of \(c\).

As another example, we consider analyzing the impact of changing \(a\). The last five transactions that involve \(a\) are \(\{a, b, c\}, \{a\}, \{a, d\}, \{a, c\}, \{a\}\). Again we use the stochastic dependency formula to compute the probabilities:

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & \{a\} & \{a, c\} & \{a, d\} & \{a\} & \{a, b, c\} & \Pr \\
\hline
b & \lambda_1 \cdot 0 & \lambda_2 \cdot 0 & \lambda_3 \cdot 0 & \lambda_4 \cdot 0 & \lambda_5 \cdot 1 & 0.0667 \\
\hline
c & \lambda_1 \cdot 0 & \lambda_2 \cdot 1 & \lambda_3 \cdot 0 & \lambda_4 \cdot 0 & \lambda_5 \cdot 1 & 0.3333 \\
\hline
d & \lambda_1 \cdot 0 & \lambda_2 \cdot 0 & \lambda_3 \cdot 1 & \lambda_4 \cdot 0 & \lambda_5 \cdot 0 & 0.2667 \\
\hline
\end{array}
\]

\(\text{As a performance consideration, we could directly ignore } d \text{ because it does not occur in any transactions with } c.\)
Using the same minimum threshold of 0.5, we would expect the change to \( a \) to not impact any other elements in this system.

### 6.2.4 Hybrid Stochastic Dependencies

When an element appears in less than \( k \) transactions in its history, we can augment the historical data with structural coupling information to compute stochastic dependencies. Recent studies \cite{143,199} show that, for some software systems, when revision history is limited, structural-based impact analysis can outperform history-based impact analysis. Based on this observation, we define a sub-family of stochastic dependencies called *hybrid stochastic dependencies* (HSDs): if \( 0 \leq \gamma \leq k \) transactions are available for computing stochastic dependencies, we let the available transactions be the \( \gamma \) most recent states in the Markov process and fill the remaining \( k - \gamma \) states with a structural coupling measure (e.g. Coupling Between Objects \cite{57}). Given \( \gamma \) transactions of revision history, a hybrid stochastic dependency is defined as:

\[
\Pr \left( X_\tau^{(a,b)} = 1 \mid X_{\tau-1}^{(a,b)} = j_1 \land \cdots \land X_{\tau-\gamma}^{(a,b)} = j_\gamma \right) = \sum_{i=1}^{\gamma} \lambda_i X_{\tau-i}^{(a,b)} + \sum_{i=\gamma+1}^{k} \lambda_i \delta(b,a)
\]

where

\[
\lambda_i \in [0,1]
\]

\[
\delta : E \times E \rightarrow [0,1]
\]

and

\[
\sum_{i=1}^{k} \lambda_i = 1
\]

In this model, \( \delta \) is a measure of \( b \)'s coupling to \( a \) from a structural coupling metric. Existing approaches \cite{28,121,122,199} that combine evolutionary dependencies with syntactic dependencies often simply take the intersection or union of the dependencies. Our approach weighs the two dependencies when combining them, thus providing a more flexible framework.

We refer to the sub-family of stochastic dependencies that incorporate structural coupling as *hybrid stochastic dependencies* (HSDs) and those that do not use structural dependencies as *pure stochastic dependencies* (PSDs). These models show that HSDs are a generalization of PSDs. Each PSD is precisely defined by the pair \( \langle \lambda, k \rangle \) while each HSD is precisely defined by the triple \( \langle \lambda, k, \delta \rangle \).

In the next section, we evaluate the accuracy of IA using several stochastic dependency types.
6.3 Evaluation

To evaluate the effectiveness of using stochastic dependencies to conduct change impact analysis, we compare the accuracy of IA using 16 members of the stochastic dependency family with different combinations of $k$ (length of history) and $\lambda$ (smoothing function). We also compare the results with two traditional structure-based IA techniques and the prevailing evolutionary-coupling-based IA techniques. Our evaluation aims to answer the following questions:

Q1: Are stochastic-based IA approaches more accurate than traditional structure-based IA techniques? We compare the accuracy of each stochastic dependency type to the accuracy of structure-based techniques for each subject system.

Q2: Are stochastic-based IA approaches more accurate than prevailing history-based IA techniques, which do not take temporal effects into consideration? As we mentioned before, the prevailing evolutionary coupling definition is a special case of our stochastic dependency family, and we conduct multiple dimensional comparisons in a similar way to answering Q1.

Q3: How does the selection of $k$ affect the accuracy of impact analysis? We investigate the hypothesis that the IA will achieve peak performance with a particular choice of $k$ and that too long or too short of a history both will negatively impact the IA performance.

Q4: How does the selection of a decreasing $\lambda$ sequence affect the accuracy of impact analysis? We investigate the hypothesis that, in general, the usefulness of a transaction decreases over time. We consider that the hypothesis is true if we observe increasing IA performance with a decreasing $\lambda$.

In this section, we first describe the software systems to which we applied our approach. Then we describe the evaluation procedure and present the results.

6.3.1 Subjects

We select five open source projects with different sizes and from different domains as the subjects of our evaluation.

Log4J Log4J$^5$ is a popular logging framework for the Java programming language.

Hadoop Common Hadoop is a Java-based distributed computing framework; Hadoop Common$^6$ provides the shared components and functionality used by other Hadoop sub-projects.

---

$^5$http://logging.apache.org/log4j/
$^6$http://hadoop.apache.org/common/
JEdit JEdit\textsuperscript{7} is a text editor that supports syntax highlighting of source code and the ability to write macros in various scripting languages.

Apache Ant Ant\textsuperscript{8} is an automated software build tool for Java programs.

Eclipse JDT Eclipse JDT\textsuperscript{9} is an abstract syntax tree analysis toolkit in the Eclipse IDE.

Table 6.2 shows some basic information of the systems that we analyzed.

<table>
<thead>
<tr>
<th>Subject</th>
<th># Releases</th>
<th>Repository Dates</th>
<th>KSLOC</th>
<th># Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log4J</td>
<td>11</td>
<td>12/14/00 – 6/10/07</td>
<td>10–15</td>
<td>3550</td>
</tr>
<tr>
<td>Hadoop Common</td>
<td>20</td>
<td>2/3/06 – 12/18/09</td>
<td>13–36</td>
<td>9319</td>
</tr>
<tr>
<td>JEdit</td>
<td>12</td>
<td>9/2/01 – 2/19/10</td>
<td>63–98</td>
<td>17339</td>
</tr>
<tr>
<td>Apache Ant</td>
<td>20</td>
<td>1/13/00 – 8/5/10</td>
<td>9–125</td>
<td>14554</td>
</tr>
<tr>
<td>Eclipse JDT</td>
<td>23</td>
<td>5/2/01 – 7/15/10</td>
<td>137–278</td>
<td>61110</td>
</tr>
</tbody>
</table>

6.3.2 Evaluation Procedure

For each subject system, we extracted transaction information from its Subversion (SVN) repository. Since Eclipse JDT uses CVS instead of SVN, we used the \texttt{cvs2svn}\textsuperscript{10} tool to convert the repository to SVN and obtain transaction information. For each transaction in revision history, we performed impact analysis on it using one of the files as the starting impact set. Assuming each transaction is a single change task, an ideal IA approach would be able to predict all files in the transaction as being in the impact scope. We perform IA on each transaction up to five times (fewer if the transaction has fewer than five files), with a different random starting impact file each time. The same starting impact files for each transaction are used for all the approaches for unbiased comparison. Consistent with the work of Zimmermann et al. [205] (and our evaluation approach from Chapter 5), we ignore transactions with more than 30 files as they are unlikely to be meaningful. These large blank operations on the code base are often changes licensing information that needs to be updated in the comment section of all source files; or other trivial housekeeping activities on the code base.

We use the standard information retrieval measures of \textit{precision}, \textit{recall}, and \(F_1\) for assessing the accuracy of IA. Precision measures how much of the predicted impact scope is correct, while recall measures how much of the actual impact scope was predicted. The \(F_1\) measure/score combines pre-

\textsuperscript{7}http://www.jedit.org
\textsuperscript{8}http://ant.apache.org
\textsuperscript{9}http://www.eclipse.org/jdt/
\textsuperscript{10}http://cvs2svn.tigris.org/
cision and recall into a single number for comparison. Next we introduce the existing IA techniques against which we compare our stochastic-based IA technique.

\[
\begin{align*}
\text{precision} & = \frac{\# \text{ correct}}{\# \text{ predicted}} \\
\text{recall} & = \frac{\# \text{ correct}}{\text{transaction size}} \\
F_1 & = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\end{align*}
\]

**Structure-based Impact Analysis**

To answer the first evaluation question, we compare our stochastic IA accuracy with that of two traditional structure-based IA techniques. The first is from the work of Briand et al. [39]. They proposed a structural coupling measure that combines whether classes from different inheritance hierarchies interact (CBO'), whether a class (directly or indirectly) aggregates another class (INAG), and the number of method calls between classes (PIM). To ease analysis, we normalize their coupling measure into the range of \([0, 1]\) by dividing by the largest measure value. In this section, we refer to this measure as the *static* dependency approach. The other structure-based approach we compare against is the work of Robillard [163]. Robillard defines an algorithm that, given a starting impact set, assigns a weight in the range \([0, 1]\) to other software elements based on analyzing the structure/topology of the a dependency graph. In this section, we refer to this measure as the *topology* dependency approach.

To perform IA on a starting impact set using these measures, we first compute the dependency value based on the software structure in the most recent release. Given a minimum threshold value, we select all files whose dependency value is at least the threshold value to be included in the impact scope. Prior to performing IA on a given software release, we first identify the minimum threshold value that maximizes the \(F_1\) measure. Essentially, we compare against the best accuracy of these structure-based IA.

**History-based Impact Analysis**

To answer the second evaluation question, we compare the IA accuracy of stochastic dependency against that of traditional evolutionary dependencies [202, 205]. We perform history-based IA on a transaction by analyzing all transactions from the beginning of the revision history to the most recent
release (in order to be fair to the structural dependency experiments). Similarly to the structural dependency experiments, we determine the best minimum support and confidence values prior to processing the transactions for each software release.

**Stochastic-based Impact Analysis**

We use the four $\lambda$ sequences presented in Table 6.1 as the smoothing function of each stochastic dependency family member. For each $\lambda$ sequence, we use the values of 5, 10, 20, and 40 for $k$ (maximum number of transactions to consider). As a result, we consider 16 members of the stochastic dependency family in total. Like with evolutionary dependencies, we consider the transaction for the most recent software release to be the latest transaction available for analysis. Although various rounding techniques are possible for our stochastic dependency value to determine whether a file is in the impact scope, we follow the same strategy as with the structural dependencies—given a minimum threshold, a file is considered in the impact scope if its stochastic dependency value is at least at large as the threshold. Also similar to the structural dependency approaches, we find the best minimum threshold for each software release.

**6.3.3 Results**

Tables 6.5, 6.6, 6.7, 6.8, and 6.9 are the results of performing IA using the 16 stochastic dependency members on the five subjects. Table 6.3 shows the average $F_1$ scores for each subject system using the history-based (column *Evolution*), topology-based (column *Topology*), static dependency-based (column *Static*), and stochastic-based (with $k = 10$) IA techniques respectively. The shaded cell in each row highlights the approach with the best average $F_1$ score over all the software releases. \(^{11}\)

Table 6.3: Stochastic Dependency Evaluation Results

<table>
<thead>
<tr>
<th>Subject</th>
<th>Average $F_1$ Score</th>
<th>Evolution</th>
<th>Topology</th>
<th>Static</th>
<th>Stochastic ($k = 10$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Const</td>
<td>Line</td>
<td>Sine</td>
<td>Expon</td>
</tr>
<tr>
<td>Log4J</td>
<td>0.3641</td>
<td>0.3082</td>
<td>0.2992</td>
<td>0.3492</td>
<td>0.3913 0.3974 0.3935</td>
</tr>
<tr>
<td>Hadoop Common</td>
<td>0.5658</td>
<td>0.5163</td>
<td>0.5174</td>
<td>0.5534</td>
<td>0.5761 0.5780 0.6034</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.3796</td>
<td>0.3525</td>
<td>0.3525</td>
<td>0.3950</td>
<td>0.3993 0.3951 0.3943</td>
</tr>
<tr>
<td>Apache Ant</td>
<td>0.4265</td>
<td>0.3733</td>
<td>0.3701</td>
<td>0.4203</td>
<td>0.4690 0.4691 0.4662</td>
</tr>
<tr>
<td>Eclipse JDT</td>
<td>0.3788</td>
<td>0.3503</td>
<td>0.3503</td>
<td>0.4764</td>
<td>0.4764 0.4764 0.4764</td>
</tr>
</tbody>
</table>

\(^{11}\)All four stochastic dependency types produced basically the same $F_1$ value for Eclipse JDT and they were all higher than the other dependencies’ results, so they are all shaded.
Q1: Comparing with Structure-base IA

These tables show that the accuracy of both structure-based IA (columns Topology and Static) for all the subjects are lower than any member of the stochastic-based IA approaches, and are also lower than traditional history-based impact analysis. We use Apache Ant and Hadoop Common as examples to elaborate more details.

For Apache Ant, we ran a total of 4306 impact analyses on 1313 transactions with each dependency type. We compute the average $F_1$ score for each dependency type, by adding the $F_1$ score for each of the 4306 IA analysis results and dividing the total by 4306. Figure 6.1 shows the average $F_1$ score for each release of Ant, using several dependency types. The figure shows that using sinusoidal stochastic dependency with $k = 10$ achieves highest accuracy of all the other dependency types.

![Figure 6.1: Average $F_1$ per Release: Apache Ant](image)

For Hadoop Common, we ran a total of 5700 impact analyses on 2633 transactions with each dependency type. In total over all five subjects, we ran 103,416 impact analyses on 47,331 transactions. Figure 6.2 shows the average $F_1$ score for each release of Hadoop, using several dependency types. This figure shows that using the stochastic dependency (with exponential $\lambda$ sequence and a $k$ value of 10 achieves) consistently outperforms the other IA approaches, often significantly.
Figure 6.2: Average $F_1$ per Release: Hadoop Common

Q2: Comparing with Traditional History-based IA

Table 6.3 shows that stochastic dependencies often outperform the traditional evolutionary dependencies for IA. To confirm our intuition based on the average $F_1$ measures, we apply the Wilcoxon signed rank test [198] to compare each of the stochastic dependency types against the traditional evolutionary dependency. The null hypothesis $H_0$ for our statistical test is that the $F_1$ value (per impact analysis on each transaction) achieved by the evolutionary dependency is the same as the $F_1$ value achieved by the stochastic dependency. The alternative hypothesis $H_1$ is that the stochastic dependency achieves greater $F_1$ value than the evolutionary dependency.

Table 6.4 shows the $W$-score and the $p$-value for each stochastic dependency type. The final column of the table denotes if the statistical test shows that the stochastic dependency is significantly more accurate at IA than the traditional evolutionary dependency. For example, when comparing the linear stochastic dependency against the traditional history-based approach, we get a $W$-score of 14404088 and a $p$-value of $< 2.2 \times 10^{-16}$, which is statistically quite significant. As we see from the table, for all the subject systems, at least one of the stochastic dependency IA approaches outperforms the evolutionary dependency approach. In the case of Eclipse JDT, all four stochastic dependency types achieved significantly higher accuracy than evolutionary dependencies.
### Table 6.4: Wilcoxon Signed Rank Test Results ($k = 10$)

<table>
<thead>
<tr>
<th>Subject</th>
<th>Stochastic $\lambda$</th>
<th>$W$</th>
<th>$p$</th>
<th>Stochastic Better?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log4J</td>
<td>Constant</td>
<td>38457.5</td>
<td>0.9989</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Linear</td>
<td>62979.5</td>
<td>$7.133 \times 10^{-6}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td>82792.5</td>
<td>$1.57 \times 10^{-7}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Expon</td>
<td>94266.5</td>
<td>$1.552 \times 10^{-4}$</td>
<td>✓</td>
</tr>
<tr>
<td>Hadoop Common</td>
<td>Constant</td>
<td>795625.5</td>
<td>1.0</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Linear</td>
<td>1094091</td>
<td>$2.654 \times 10^{-7}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td>1131795</td>
<td>$8.406 \times 10^{-7}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Expon</td>
<td>1988273</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td>JEdit</td>
<td>Constant</td>
<td>14149902</td>
<td>$5.043 \times 10^{-6}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Linear</td>
<td>14404088</td>
<td>$1.054 \times 10^{-6}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td>14207902</td>
<td>0.001198</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Expon</td>
<td>14337290</td>
<td>0.2036</td>
<td>✓</td>
</tr>
<tr>
<td>Ant</td>
<td>Constant</td>
<td>295247</td>
<td>0.9938</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Linear</td>
<td>1209382</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td>1194846</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Expon</td>
<td>1071560</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td>Eclipse JDT</td>
<td>Constant</td>
<td>468750088</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Linear</td>
<td>468750088</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td>468750088</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Expon</td>
<td>468750088</td>
<td>$&lt; 2.2 \times 10^{-16}$</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Q3: Effects of $k$**

Reading horizontally in Tables 6.5, 6.6, 6.7, 6.8, and 6.9, we can see how the $F_1$ results change with the increase in $k$. Eclipse JDT appears to be different from other systems in that the $F_1$ score does not change with the changes to $k$. For all other four systems, the accuracy first increases with the increase of $k$ and then decreases as $k$ continues increasing. For example, in Apache Ant, using a linear $\lambda$ sequence with $k = 5$ (second row, first column in Table 6.5), we achieved an average $F_1$ of 0.4665. Increasing the length of history to $k = 10$ (second row, second column), we improved the accuracy to 0.4690. However, further increasing the length of history to 20 and 40, caused the accuracy to begin decreasing. This increase in accuracy and then decrease in accuracy from increasing $k$ was consistent in the four subject systems, but the point of decrease varied among the systems. For most of the systems, the accuracy began decreasing at $k = 10$, and for JEdit, the accuracy began to decrease at $k = 20$. These changes in accuracy allows us to positively answer evaluation question Q3: changing the amount of historical data analyzed ($k$) does affect the accuracy of stochastic-based IA.
Table 6.5: $F_1$ Scores for $\langle \lambda, k \rangle$ Pairs: Apache Ant

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$k$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td></td>
<td>0.4414</td>
<td>0.4203</td>
<td>0.4005</td>
<td>0.3905</td>
</tr>
<tr>
<td>Linear</td>
<td></td>
<td>0.4665</td>
<td>0.4690</td>
<td>0.4394</td>
<td>0.4203</td>
</tr>
<tr>
<td>Sinusoidal</td>
<td></td>
<td>0.4657</td>
<td>0.4691</td>
<td>0.4417</td>
<td>0.4203</td>
</tr>
<tr>
<td>Exponential</td>
<td></td>
<td>0.4656</td>
<td>0.4662</td>
<td>0.4662</td>
<td>0.4662</td>
</tr>
</tbody>
</table>

Table 6.6: $F_1$ Scores for $\langle \lambda, k \rangle$ Pairs: Hadoop

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$k$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td></td>
<td>0.5679</td>
<td>0.5534</td>
<td>0.5415</td>
<td>0.5289</td>
</tr>
<tr>
<td>Linear</td>
<td></td>
<td>0.5919</td>
<td>0.5761</td>
<td>0.5597</td>
<td>0.5428</td>
</tr>
<tr>
<td>Sinusoidal</td>
<td></td>
<td>0.5922</td>
<td>0.5780</td>
<td>0.5608</td>
<td>0.5443</td>
</tr>
<tr>
<td>Exponential</td>
<td></td>
<td>0.6005</td>
<td>0.6034</td>
<td>0.6032</td>
<td>0.6032</td>
</tr>
</tbody>
</table>

**Q4: Effects of $\lambda$**

Reading vertically in these tables, we can see how the $\lambda$ sequences affect accuracy. For example, in Apache Ant, with a constant $\lambda$ and $k = 5$, we achieved an average $F_1$ of 0.4414. Keeping the same length of history, but switching to a sinusoidal $\lambda$ sequence, we increased the accuracy to 0.4657. Switching to a linear sequence slightly increased the accuracy to 0.4665. For Apache Ant, the linear and sinusoidal sequences both achieved similar accuracies with the best combination being a sinusoidal sequence and 10 transactions (shown as a shaded cell in the table). This trend is consistent with all the system except for Eclipse JDT, where the changes did not make a difference.

Among the other four systems we evaluated, the decreasing $\lambda$ sequences always outperformed the constant $\lambda$ sequence. Hence, we can positively answer evaluation question Q4: using a decreasing $\lambda$ sequence to emphasize more recent historical data does often improve the accuracy of stochastic-based IA.

At first, we find it unusual that neither the selection of history length or $\lambda$ affects the accuracy for Eclipse JDT. We hypothesize that the reason for this is due to the stability of the JDT architecture.

Table 6.7: $F_1$ Scores for $\langle \lambda, k \rangle$ Pairs: JEdit

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$k$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td></td>
<td>0.3775</td>
<td>0.3950</td>
<td>0.3954</td>
<td>0.3820</td>
</tr>
<tr>
<td>Linear</td>
<td></td>
<td>0.3892</td>
<td>0.3993</td>
<td>0.4096</td>
<td>0.3990</td>
</tr>
<tr>
<td>Sinusoidal</td>
<td></td>
<td>0.3890</td>
<td>0.3951</td>
<td>0.4056</td>
<td>0.3975</td>
</tr>
<tr>
<td>Exponential</td>
<td></td>
<td>0.3935</td>
<td>0.3943</td>
<td>0.3943</td>
<td>0.3943</td>
</tr>
</tbody>
</table>
Table 6.8: $F_1$ Scores for $(\lambda, k)$ Pairs: Log4J

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$k$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>0.3712</td>
<td>0.3492</td>
<td>0.3357</td>
<td>0.3237</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>0.3927</td>
<td>0.3913</td>
<td>0.3752</td>
<td>0.3391</td>
<td></td>
</tr>
<tr>
<td>Sinusoidal</td>
<td>0.3937</td>
<td>0.3974</td>
<td>0.3724</td>
<td>0.3379</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>0.3904</td>
<td>0.3935</td>
<td>0.3935</td>
<td>0.3935</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.9: $F_1$ Scores for $(\lambda, k)$ Pairs: Eclipse

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$k$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td></td>
</tr>
<tr>
<td>Sinusoidal</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td>0.4764</td>
<td></td>
</tr>
</tbody>
</table>

If a design is highly stable, then the same impact scope (i.e., the same transactions) occur repeated throughout the history and taking any slice of the history would yield similar data. The similar data causes our tool to infer similar stochastic dependency values and hence highly similar average $F_1$ scores. This stable architecture and the similarity of history can also explain why the different $\lambda$ sequences produce similar accuracies. Further investigation of this hypothesis regarding the stability of Eclipse JDT remains a future work.

6.3.4 Hybrid Stochastic Dependencies

Finally, we perform a preliminary evaluation of HSDs to see if using structural coupling information can improve IA accuracy. To address this evaluation question, we slightly modify the evaluation framework to limit the amount of historical data used for analysis, since the effects of HSDs can diminish with extensive history. To perform IA on a transaction $i$, we only consider (at most) the $k$ transactions prior to $i$ that occur after the most recent release. For example, if transaction #200 is a release of the software system and we want to perform IA on transaction #206 (with $k = 10$) then we only use transactions 201–205 for the Markov chain. Additionally, we only look at the first 40 transactions per release, since later transactions may not use the structural dependency in HSDs.

For brevity, we only detail the results for Apache Ant. The results for the other subject systems are similar to those of Ant. Table 6.10 shows the average $F_1$ values for the HSD comparison of Ant. We note that, due to the difference in evaluation, the $F_1$ values in Table 6.10 cannot be compared to the values in Table 6.5. Two different hybridizations of stochastic dependencies were evaluated,
using the two structural coupling measures discussed earlier in this section. We see from the table that HSDs consistently outperform PSDs when historical data is limited. Additionally, the table shows the optimal type of structural dependency used for hybridization varies based on the original PSD used (and the subject system). However, although HSDs display a slight advantage to PSDs, this improvement in accuracy is minimal in our evaluation. While using HSDs may offer significant benefits over PSDs when revision history data is limited, for the subject systems we evaluated and the structural dependencies we used, the results are not convincing enough for immediate practical use. In Section 6.4.2, we discuss possible future work to continue exploring the use of HSDs.

Table 6.10: Hybrid Stochastic Dependencies Results: Apache Ant

<table>
<thead>
<tr>
<th>λ Type</th>
<th>k</th>
<th>Pure</th>
<th>+ Topology</th>
<th>+ Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>10</td>
<td>0.4002</td>
<td>0.3998</td>
<td>0.4010</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.3738</td>
<td>0.3766</td>
<td>0.3757</td>
</tr>
<tr>
<td>Linear</td>
<td>10</td>
<td>0.4403</td>
<td>0.4412</td>
<td>0.4393</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.3918</td>
<td>0.3928</td>
<td>0.3902</td>
</tr>
<tr>
<td>Sinusoidal</td>
<td>10</td>
<td>0.4423</td>
<td>0.4447</td>
<td>0.4445</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.3928</td>
<td>0.3939</td>
<td>0.3937</td>
</tr>
<tr>
<td>Exponential</td>
<td>10</td>
<td>0.4327</td>
<td>0.4337</td>
<td>0.4339</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.4327</td>
<td>0.4337</td>
<td>0.4339</td>
</tr>
</tbody>
</table>

6.4 Discussion

This section discusses some threats to validity to our framework and evaluation. We also describe some possible future work with stochastic dependencies.

Although evolutionary dependencies are a type of stochastic dependencies that use a constant λ sequence, not all stochastic dependencies with constant λ sequences are evolutionary dependencies. For a stochastic dependency type to predict the same impact scope as evolutionary dependencies, we need to have a constant λ sequence and k must be the number of transactions that involves the the starting impact set. That is why, in our evaluation, the stochastic dependency types with a constant λ sequence did not perform the same as traditional logic-based IA. To achieve the same results as logic-based IA, we would have needed to vary the value of k for each transaction.

The low accuracy of all the dependency types may seem alarming at first (the average $F_1$ measure dips down to 0.3). However, the accuracy in actual usage may be significantly higher. This low accuracy in the evaluation is due to two factors: the directional nature of change impact and the selection of a random starting impact set. As an example, we consider a transaction that contains
files A, B, and C—where B and C subclasses of A. By the directional nature of change impact, we mean that changing A affects its subclasses B and C. However, changing B or C does not affect the parent class A. Since we do not know that A is the correct starting impact set, our evaluation approach considers each file to be a starting impact set and expects the IA approach to identify the other two files. Hence, even if an IA approach correctly identifies the proper impact scopes, it would only achieve a 0.33 average $F_1$ score ($F_1 = 1$ with A as starting impact set, $F_1 = 0$ with B as starting impact set, and $F_1 = 0$ with C as starting impact set). Identifying the actual starting impact set of a transaction or modification request is an active, but immature, area of research (e.g. Kagdi et al. [123], Antoniol et al. [12]), and can be used to improve IA evaluation in the future.

6.4.1 Threats to Validity

Since we only applied our approach to five Java-based, object-oriented subject systems, we cannot conclude that the effectiveness of stochastic dependencies generalizes to all software systems; however, we did choose projects of various sizes and domains to begin addressing this issue.

As with any technique that derives dependencies from transactions in revision history, our approach assumes that the transactions represent cohesive work units. In other words, if developers only commit to the revision control system once a week, the files committed together may have no semantic relationship and are only coincidentally occurring in the same transaction. The accuracy of IA produced by stochastic dependencies and evolutionary dependencies indicate that transaction often are these cohesive work units and often do indicate semantic coupling of software elements.

Our stochastic dependency framework assumes that the transactions of a revision control history form a stochastic process where the dependencies between software elements control the probability distribution of the random variables in this process. We use this assumption as the basis for building the Markov chain model. In reality, the probability distribution of these random variables (software elements that occur together in a transaction) can also be influenced by external factors and our assumption may not always be true. However, our evaluation shows that modeling the revision history as a stochastic process is a sufficient approximation for the actual behavior of development to yield accurate impact analysis results.

We only compared the stochastic-based IA to several existing approaches, and there are many other types of IA. Our study does show that the temporal dimension of history data does impact the accuracy of a change impact analysis technique. We also find that systems with different evolution characteristic and stability needs different types of dependencies to effectively conduct IA.
6.4.2 Future Work

Exploring methods to improve stochastic-based IA accuracy in general is an ongoing work. In particular, we are exploring the use of more complex probabilistic models (e.g. hidden Markov models, dynamic Bayesian networks [165]) in place of the Markov chain model. Better incorporating structural dependency information into these models for IA is also of interest. Defining and evaluating other sub-families of stochastic dependencies is a future work.

The parameters of stochastic dependency allow for a large number of specific dependency types to be defined. Identifying additional, successful $\lambda$ sequences is an ongoing work. For example, a possible $\lambda$ sequence could be based on the how long ago (i.e., in terms of days, months, etc.) a transaction was committed. Exploring techniques to automatically construct $\lambda$ and $k$ values for a given software system is also a possible future work.

While efficient data structures and algorithms have been developed for mining of association rules for evolutionary dependency analysis, such techniques are not yet available for stochastic dependencies. In our evaluation, the performance of stochastic-based IA was a constant factor slower than the logic-based IA, excluding the time for preprocessing evolutionary dependencies (e.g. building the FP-tree). Developing algorithms to improve the performance of stochastic dependencies remains a future work. It is worth noting that the amount of preprocessing required by traditional evolutionary dependencies is rather expensive and can hinder its use in practice. Preprocessing is often perform per release of a software system (as with our evaluation) and the same data is used until the next release. Hence, the data used for IA may not reflect the latest changes to a system. On the other hand, stochastic dependencies can be computed on the fly without preprocessing to reflect the most current state of the system.

Software Element Ownership

While some organizations maintain information on software element ownership, many organizations (especially open source projects) do not keep track of the current owners of each element. To address the problem of unrecorded ownership, some researchers (e.g. Minto and Murphy [142]) apply evolutionary dependencies between people and elements to identify owners. The problems that plague evolutionary dependencies between software elements also affect its use in mapping people to elements. Therefore, we intend to apply the concept of stochastic dependencies to automatically link elements and owners as a future work.
Next, we describe our initial thoughts on formalizing this ownership problem, using stochastic dependencies. Let $P = \{ p_i \}_{i=1}^Q$ be the set of persons involved in the software project. We define a sequence $W = \{ w_i | 1 \leq i \leq M \land w_i \subseteq P \}$ to identify the set of people who work on each transaction $t_i$. To determine who worked on a transaction, we can look at who committed the transaction, information in credit files (if any are available), etc. Let $W^{(e)} = \{ w^{(e)}_i \in W | e \in t_i \}$ be the sequence of people who work on software element $e \in E$. For each person $p \in P$ and software element $e \in E$, we define a stochastic process $\Pi^{(e,p)} = \{ \pi^{(e,p)}_i \}_{i=1}^Q$ such that:

$$\pi^{(e,p)}_i = \begin{cases} 1 & \text{if } p \in w^{(e)}_i \\ 0 & \text{otherwise} \end{cases}$$

Using the same definition for element-to-element stochastic dependencies, we can define the people-to-element stochastic dependencies:

$$\Pr(\pi_\tau = j_0) = \Pr(\pi_\tau = j_0 | \pi_{\tau-1} = j_1 \land \pi_{\tau-2} = j_2 \cdots \land \pi_{\tau-k-1} = j_k) = \sum_{i=1}^k \lambda_i \pi_{\tau-i}$$

where $\lambda_i \in [0, 1]$ and $\sum_{i=1}^k \lambda_i = 1$

Unlike element-to-element stochastic dependencies, if fewer than $k$ transactions are available, we do not have another model of people to element dependencies that we can combine with the stochastic dependencies. Hence, if only $\gamma$ transactions are available, then we only use those $\gamma$ to compute the stochastic dependency.

6.5 Conclusion

Using history-based evolutionary coupling to conduct change impact analysis in software maintenance has gained popularity recently. The temporal dimension of the historical data influences the accuracy of such analysis because data in distant history may not be valid, especially for a system that is volatile and subject to frequent refactoring. Our stochastic dependency family formally accounts for the characteristic of evolution history. By varying the smoothing function and the length of history, a developer can select the combination of parameters that best fits the system. Our experiment
shows that performing IA using stochastic dependencies consistently outperforms structure-based IA techniques. Comparing with prevailing history-based IA techniques, our stochastic-based approach also obtains better IA accuracy for all five subject systems.
7. Identifying Modularity Violations

The essence of software modularity is to allow for independent module evolution and independent task assignment [22,156]. In reality, however, two modules that are supposed to be independent may consistently change together, due to unexpected side effects caused by quick and dirty implementation. For example, inexperienced developers may forget to remove experimental scaffolding code that should not be kept in the final product, and an application programming interface (API) may be accidentally defined using non-API classes [117]. Such activities cause modularity decay over time and may require expensive system-wide refactorings. Though empirical studies have revealed a strong correlation between software defects and eroding design structure [53,171], traditional verification and validation techniques do not find modularity violations because these violations do not always directly influence the functionality of software systems.

This chapter presents an approach that detects and locates modularity violations. Our approach compares how components should change together, based on modular structure, and how components actually change together, as reflected in the revision history. The rationale is that, if two components frequently change together to accommodate modification requests, but they belong to two separate modules that are supposed to evolve independently, we consider this as a modularity violation.

Our approach has three parts. First, we leverage the design rule hierarchy (from Chapter 5) to manifest independently evolvable modules, from which we determine structural coupling—how components should change together. Second, we mine the project’s revision history to model evolutionary coupling—how components actually change together [91]. We identify modularity violations by comparing the results of structural coupling based impact analysis (IA) with the results of evolutionary coupling based impact analysis.

We applied our approach to the version histories of two large-scale open source software systems: 15 releases of Hadoop Common,¹ and 10 releases of Eclipse JDT.² Our evaluation strategy was to identify violations for each pair of releases. If a violation was indeed problematic, it is possible that developers recognized and fixed it in a later release through a refactoring. We considered a detected violation as being confirmed if it was indeed addressed or recognized by developers later.

We used two complementary evaluation methods. First, we compared the detected violations with

¹http://hadoop.apache.org/common/
²http://www.eclipse.org/jdt/
refactorings automatically reconstructed using Kim et al.’s API matching technique [128]. Second, for the remaining detected violations, we manually examined modification requests to see whether those violations were at least recognized by developers. Because it is possible that the violations detected in recent versions are not recognized by the developers yet, we also manually examined the corresponding code to determine whether the code shows any symptoms of poor design.

We identified 231 violations from 490 modification requests of Hadoop, of which 152 (65%) violations were confirmed. From 3458 modification request of Eclipse JDT, our approach identified 399 violations, which shows that the changes in Eclipse match its modular structure better. Among these violations, 161 (40%) were confirmed. The results also show that our approach identifies modularity violations much earlier than manual identification by developers so that designers can be notified to avoid accumulating modularity decay. Third, the identified violations include symptoms of poor design, some of which cannot be easily detected using existing approaches.

The rest of this chapter is organized as follows. Section 7.1 describes a motivating example involving our approach. Section 7.2 presents related work and how our approach differs from existing approaches. Section 7.3 describes our modularity violation detection approach. Section 7.4 details our evaluation method and empirical results. Section 7.5 discusses the strengths and limitations of our approach, as well as possible future work. Section 7.6 concludes.

### 7.1 Motivating Example

In this section, we use an example to illustrate how our framework can help with modularity violation detection during software evolution. Alice is a project manager. Her team just released version $n$ of a software system. Many modification requests (MRs) were fulfilled during this release, including both bug fixes and feature enhancements. These modifications were accomplished by multiple developers with different levels of capability and experience. Before Alice launches the development process for release $n + 1$, she wants to make sure that the modular structure of the system is well-maintained. That is, she would like to check that there was no quick and dirty implementation, which introduced unexpected dependencies or broke important design rules. Fixing these problems would ensure that they do not accumulate into severe modularity decay. Our design violation detection tool, Clio, analyzes the revision history and modification requests to help Alice identify potential design problems.
As a design problem reported to Alice, consider the example shown in Figure 7.1. Class Grocery is a subclass of Item. The public interface of class Item is a design rule [22] in the sense that it hides changes to the implementation of Item and it should remain stable. Using our approach, design-rule based impact analysis would never report that changes to the implementation of Grocery will require compensating changes to its superclass Item. However, if evolutionary-coupling based prediction shows that Item frequently changes due to changes in Grocery, then a set of discrepancies will be reported to Alice showing that there is a deviation involving Grocery and Item and with high support and confidence. Such deviation is problematic because it may cause a ripple effect to other components that use Item’s interface. We suspect that changes to Grocery is violating Item’s design rule, meaning that there exists a modularity violation between them.

Figure 7.1: Class Diagram for Example Modularity Violation

7.2 Related Work

In this section, we compare and contrast our approach with other related research topics.

Automatic Detection of Bad Code Smells. Fowler [88] describes the concept of bad smell as a heuristic for identifying redesign and refactoring opportunities. Example bad smells include code clone and feature envy. Garcia et al. [93] proposed several architecture-level bad smells. To automate the identification of bad smells, Moha et al. [146] presented the Decor tool and domain specific language (DSL) to automate the construction of design defect detection algorithms. Several other techniques [191–193] automatically identify bad smells that indicate needs of refactorings. For example, Tsantalis and Chatzigeorgiou’s technique [192] identifies extract method refactoring opportunities using static slicing. Detection of some specific bad smells such as code duplication has also been extensively researched. Higo et al. [112] proposed the Aries tool to identify possible refactoring candidates based on the number of assigned variables, the number of referred variables, and dispersion in the class hierarchy. A refactoring can be suggested if the metrics for the clones satisfy certain predefined values.

Our modularity violation detection approach is different from these existing approaches in several aspects. First, it is not confined to particular types of bad smells. Instead, we hypothesize that
multiple types of bad smells are instances of modularity violations that can be uniformly detected by our approach. For example, when code clones frequently change together, our approach will detect this problem because the co-change pattern deviates from the designed modular structure. Second, by taking version histories as input, our approach detects violations that happened most recently and frequently, instead of bad smells detected in a single version without regard to the program’s evolution context. Similar to our approach, Ratzinger et al. [162] also detect bad smells by examining evolutionary coupling. Their approach leaves it to developers to identify modularity violations from visualization of evolutionary coupling, while our approach locates violations by comparing evolutionary coupling with structural coupling. The detected violations thus either reflect the problem in the original design or introduced in the subsequent modification requests.

**Design Structure Matrix Analysis.** Some of the most widely used design structure matrix (DSM) tools are Lattix, Strutere 101, and NDepend. These tools support automatic derivation of DSMs from a code base, in which columns and rows model classes or files, and the dependencies model function calls, inheritance, etc. Different from these tools, the DSMs used in our approach are generated from augmented constraint network (ACN) models, which separate the interface and implementation of a class into two design dimensions, and manifest implicit and indirect dependencies. As Chapter 4 showed, an ACN-derived DSM can capture more dependencies than that of a syntactical DSM.

Sangal et al. [166] identify certain types of modularity violations using Lattix DSMs. Using Lattix, the user can specify which classes should not depend on (i.e., syntactically refer to) which other classes. The tool notifies the user if such predefined constraints are violated. A key difference between our approach and the Lattix violation detection technique is that our approach uses version histories as opposed to analyzing a single version only. Our approach detects violations that occur during software evolution, many of which are not in the form of syntactical dependency, and thus will not be detected by Lattix. Another major difference is that our approach takes recency and frequency into consideration when identifying modularity violations.

**Dependency Structure and Software Defects.** The relation between software dependency structure and defects has been widely studied. Many empirical evaluations (e.g., Selby and Basili [171], Cataldo et al. [53]) have found that modules with lower coupling are less likely to contain defects than

---

3http://www.lattix.com/
4http://www.headwaysoftware.com/products/structure101/
5http://www.ndepend.com/
those with higher coupling. Various metrics have been proposed (e.g., Chidamber and Kemerer [57]) to measure coupling and failure proneness of components. The relation between evolutionary coupling [91] and defects has also been recently studied. Cataldo et al.’s [53] study revealed a strong correlation between density of evolutionary coupling and failure proneness. Fluri et al.’s [87] study shows that a large number of evolutionary coupling relationships are not entailed by structural dependencies. While the purpose of these studies are to statistically account for the relationship between software defects, evolutionary couplings, and syntactic dependencies, the purpose of our approach is to locate modularity violations that may cause design decay and software defects.

7.3 Approach

In this section, we first present an overview of our modularity detection framework. Then we detail each part of the approach, as implemented in our Clio tool.

7.3.1 Framework Overview

Figure 7.2 depicts an overview of our approach that takes the following artifacts as input. The approach is implemented by our Clio tool, as a framework of pluggable components (plugins). The first input is the original modular structure of version \( n \) before implementing these modification requests. Since an accurate design model is usually not available in practice, Our approach uses the Moka tool to reverse-engineer UML class diagrams from compiled Java binaries. Our Clio tool then uses the Janus tool to transform a class diagram into an augmented constraint network (ACN) [46]. From an ACN, a design structure matrix (DSM) can be automatically derived [46, 50].

The second input is the revision history of the project, which is used to derive evolutionary coupling from a set of files changed to implement modification requests. The extrapolis plugin of Clio computes evolutionary coupling at a file level (following the technique of Ying et al. [202]) and saves it to a data store for later use.

The third input is the detailed information about a set of files \( S \) (called the MR solution), which was modified to fulfill each modification request.

For each modification request, the dr-predict plugin outputs the components that are likely to be changed according to the original modular structure (FileSet A in Figure 7.2). Clio’s evol-predict plugin also reports the components that are likely to be changed according to evolutionary coupling, recorded in FileSet B. Finally, given \( A \) and \( B \), and a MR solution \( S \), the detect plugin computes a set
of discrepancies, \( D = (B \cap S) \setminus A \). By using \( B \cap S \), we filter out files that were accidentally changed together. Recurring discrepancies (a subset of files in \( D \)) are reported to the users as violations.

\[
\text{Compiled Binaries} \rightarrow \text{Moka} \rightarrow \text{Janus} \rightarrow \text{dr-predict Plugin}
\]

\[
\text{Revision History} \rightarrow \text{extrapolis Plugin} \rightarrow \text{Evolution Deps} \rightarrow \text{evol-predict Plugin}
\]

\[
\text{Modification Request: S} \rightarrow \text{File Set: A} \rightarrow \text{File Set: B} \rightarrow \text{detect Plugin}
\]

\[
\text{Tool} \rightarrow \text{Artifact} \rightarrow \text{Database} \rightarrow \text{Discrepancy: D}
\]

Figure 7.2: Modularity Violation Identification Approach Overview

In the following subsections we discuss each of the Clio plugins in detail. First, we describe the extrapolis and evol-predict plugins that perform impact analysis based on evolutionary coupling. Then, we explain our modularity-based impact analysis technique as implemented in the dr-predict plugin. Finally, we explain how the detect plugin uses the impact analysis results to identify modularity violations.

7.3.2 History-based Impact Analysis

Since performing impact analysis using evolutionary dependencies requires a preprocessing of the revision history, our extrapolis plugin performs this preprocessing on the revision history. It extracts evolutionary coupling between files, storing the support and confidence values between files into a database following Ying et al. [202] and Zimmermann et al. [204]. The evol-predict plugin then reads the coupling information from the database and identifies the impact scope (noted as FileSet B in Figure 7.2) from the starting impact set (described below). A file is predicted to be in the impact
scope if the corresponding association rule’s support and confidence are above the minimum support \( \sigma \) and confidence \( \chi \) thresholds.

For each modification request, the \textit{evol-predict} plugin first selects a subset of files in the corresponding change set (solution) that exhibit the strongest evolutionary coupling according to the preprocessed data. We call this selected set of files, the starting impact set \( \psi \), as the discrepancy between \( \psi \)’s impact scope based on structural coupling and \( \psi \)’s impact scope based on evolutionary coupling is mostly likely to reveal modularity violations.

7.3.3 Modularity-based Impact Analysis

The modularity-based impact analysis of our framework is implemented by the \textit{dr-predict} plugin. This plugin takes, as input, a design rule hierarchy (DRH) of the design and the starting impact set \( \psi \), which was identified by the \textit{evol-predict} plugin. From the DRH graph and starting impact set, the \textit{dr-predict} plugin analyzes \( \psi \)’s impact scope to identify the components that should change together according to the modular structure if \( \psi \) changed. We leverage Robillard’s [163] relevant artifact recommendation algorithm, which identifies a subset of nodes in a graph, relevant to an initial set of interests based on the graph’s topology. We chose this algorithm because the input format and the algorithm requirements are similar: (1) a DRH is a directed graph just like a static dependency graph in Robillard’s and (2) the algorithm must carefully propagate the degree of relevance (weights) along edges until they are stabilized using an iterative, fix-point algorithm.

In order to demonstrate our modularity-based change scope analysis approach, we depict a small subset of the maze game DRH graph in Figure 7.3 for the purpose of illustration. In Figure 7.3, we only show 1 of the 2 modules in layer 1, 3 modules each from layer 2 and 3, and 1 module from layer 4. We note that the edges of the DRH graph are populated based on constraints in the ACN as introduced in Chapter 5.

The vertices with shaded background and white text model the starting impact set from an MR. Beginning with the starting impact set, we assign a weight \( \mu \), in the range \([0, 1]\), to each vertex, in a breadth-first order. The starting impact set vertices are assigned a weight of 1 and added to a initial set of interests, \( S \). From vertex \texttt{Room Interface}, we examine its neighbors, the subordinating decisions that \texttt{Room Interface} influences, and assign them each a weight. While traversing the graph to assign weights, we ignore the starting change set’s design rules to ensure that they remain stable. For example, since the \texttt{Room} class is the starting change set in our example, then its design rules, \texttt{MapSite}’s interface and implementation should not be within the impact scope.
Robillard [163] defines a formula for computing the weight of a vertex:

$$\mu_0 = \left( \frac{1 + |S_{\text{forward}} \cap S|}{|S_{\text{forward}}|} \cdot \frac{|S_{\text{backward}} \cap S|}{|S_{\text{backward}}|} \right)^\alpha$$

Using this formula, we assign higher weights to vertices that share more edges with elements in the set of interest $S$. This allows us to identify the components that are likely to be affected by the starting change set due to the strengths of their design-level dependencies. $\mu$ is a weight and $\alpha$ is a constant defined to determine the degree of relevancy propagation. Following the results of Robillard, we use $\alpha = 0.25$ for our evaluation. However, the value of $\alpha$ does not change the order of suggested elements.

To start the each iteration of the algorithm, we take all the vertices that have just been assigned weights, add them to the set of interest $S$, and use them as the starting points for weight assignment. We repeat this process of iteratively assigning weights to vertices until the new weights fall below a certain threshold. All vertices that were not assigned a weight are considered to have the minimum weight of 0. Figure 7.3 shows the weights for each vertex after all weights are propagated. The vertices whose weights are above the threshold $m$ (e.g. 0.75) are then recommended as being in the impact scope (noted as node with dotted circles). Below, we discuss how this minimum threshold $m$ is determined.
7.3.4 Discrepancy Analysis

For each software release, we vary the thresholds to find values that maximize measure of accuracy over all the MRs. These threshold values are used for impact analysis in the next release of the software system. By varying the threshold values, we continually optimize the IA approaches to be accurate. With \textit{dr-predict}, we vary the minimum weight threshold $m$ from 0 to 0.95 in increment of .05. With \textit{evol-predict}, we independently vary $\sigma$ from 2 to 10 and $\chi$ from 0 to 0.95 in increment of .05. To measure accuracy of impact analysis, we use the standard $F_1$ measure from information retrieval. $F_1$ is computed based on the measures of \textit{precision} and \textit{recall}, as defined below:

$$\text{precision} = \frac{\text{# correct}}{\text{# predicted}}$$

$$\text{recall} = \frac{\text{# correct}}{\text{solution size}}$$

$$F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

For each MR, we identify discrepancies between modularity-based impact scope and history-based impact scope. We then compute recurring discrepancies over several versions of the software by using a frequent-pattern mining algorithm [101]. The recurring patterns among these discrepancies are called \textit{modularity violations}. For example, we consider two MRs with the same starting impact set of $a$. Suppose that the set of discrepancies is $\{\{a,b,c\}, \{a,b\}\}$. Then, we say that $\{a,b\}$ is a modularity violation that occurred twice, and $\{a,b,c\}$ is a modularity violation that occurred once.

For example, \texttt{EnchantedMazeFactory_impl} and \texttt{BombedMazeFactory_impl} are both located in the last layer of the DSM, meaning that they should evolve independently from each other. The \textit{dr-predict} plugin will never report that they are in each other’s change scope. If the revision history shows that they consistently change together (e.g. due to similar changes to cloned code), CLIQ will report that there is a modularity violation. We consider another example: since \texttt{Mapsite\_interface} is the design rule of \texttt{Room\_impl}, it is normal that \texttt{Mapsite\_interface} changes and influences \texttt{Room\_impl} along with other dependent components. But CLIQ’s \textit{dr-predict} plugin will never predict \texttt{Mapsite\_interface} to be within the change scope of \texttt{Room\_impl}. However, if the revision history shows that whenever \texttt{Room} changes, \texttt{MapSite} always changes with it, it is a violation because all other components that depend on \texttt{MapSite} may be affected, causing unwanted side effects.
7.4 Evaluation

To assess the effectiveness of our modularity violation detection approach, the evaluation aims to answer the following questions:

Q1. How accurate are the modularity violations identified by our approach? That is, do these identified violations indeed indicate problems? Given the difficulty of finding the designers of the subject systems who can accurately answer this question, we evaluate our approach retrospectively and conservatively: we examine the project’s version history to see whether and how many violations we identified in earlier versions are indeed refactored in later versions or recognized as design problems by the developers (e.g. through modification requests, source code comments). The precision calculated this way is the most conservative, lower-bound estimation because it is possible that some violations we identified have not been recognized by the developers yet, and could be refactored in future releases. We do not calculate the recall of our result because it is not possible to find all possible design issues in a system.

Q2. How early can our approach identify problematic violations? Our purpose is to see if this approach can detect design problems early in the development process. Although it may not be necessary to fix a violation as soon as it appears, revealing these violations will make the designers aware as soon as possible to avoid accumulating modularity decay. For each confirmed violation, we compare the version where it was identified with where it was actually refactored or recognized by the developers.

Q3. What are the characteristics of design violations identified by our approach? We also examined the detected violations’ corresponding code to see whether they show any symptoms of poor design and categorized the violations into four categories.

7.4.1 Subjects

We choose two large-scale open source projects, Hadoop Common and Eclipse Java Development Tools (JDT) as our evaluation subjects. Hadoop is a Java-based distributed computing framework. We applied our approach to the first 15 releases, 0.1.0 to 0.15.0, covering about three years of development. Eclipse JDT is the core AST analysis toolkit in the Eclipse IDE. We studied 10 releases of Eclipse JDT, from release 2.0 to 3.0.2, also covering about three years of development.
Our evaluation use both their revision histories and source code. For Hadoop, we investigated their SVN repository to extract transactions. Eclipse JDT used CVS instead of SVN, so we use the\_cvs2svn\textsuperscript{6} tool to derive the transactions. In Table 7.1, we present some basic data regarding to Hadoop and Eclipse JDT that we studied. We removed commits with only one file or more than 30 files because they either do not contribute to modularity violation detection or they include noise such as changes to license information. For each release pair \( n \) and \( n + 1 \), we computed discrepancies between the results of structural-coupling based impact scope analysis and the results of change-coupling based impact scope analysis. We then accumulated the discrepancies over the five most recent releases to identify recurring violations that occur more than a certain number of times. The experiments showed that the results do not significantly change if we aggregate discrepancies over more than five releases.

### Table 7.1: Characteristics of Modularity Violation Detection Subjects

<table>
<thead>
<tr>
<th>Subjects</th>
<th>KSLOC</th>
<th># Transactions</th>
<th># Releases</th>
<th># MRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse JDT</td>
<td>137–222</td>
<td>27806</td>
<td>10</td>
<td>3458</td>
</tr>
<tr>
<td>Hadoop Common</td>
<td>13–64</td>
<td>3001</td>
<td>15</td>
<td>490</td>
</tr>
</tbody>
</table>

#### 7.4.2 Evaluation Procedure

We ran our experiments on a Linux server with two quad-core 1.6Ghz Intel Xeon processors and 8GB of RAM. We evaluate the output of our Clio tool (i.e., a set of violations) by checking the source code and MR records of later versions to see if they were indeed refactored or recognized as having a design problem. If so, we call such violation as being confirmed. We use both automated method and manual inspection to confirm a violation.

First, we compared the detected violations with refactorings that were automatically reconstructed by Kim et al.’s API matching tool \[128\]. This API matching tool takes two program versions as input and detects nine different types of refactorings at a method-header level (e.g. class rename, method signature change). This tool extracts method-headers from both old and new versions, finds a set of seed matches based on name similarity, generates candidate high-level transformations based on the seed matches, and iteratively selects the most likely high-level transformation to find a set of method-header level refactorings. We chose this technique because it has a 5.01% higher precision than other similar techniques according to a recent comparative study \[200\].

\textsuperscript{6}\url{http://cvs2svn.tigris.org/}
As these automatically reconstructed refactorings are method-header level refactorings, we aggregate them up to a class level to compare with the violations our approach identifies. For each violation that is matched with a reconstructed refactoring, we manually checked the refactoring to verify that it was indeed a correct refactoring that fixes design problems since the API-matching tool can report false positive refactorings. We consider a violation as confirmed if it overlaps with a valid class-level refactoring.

Second, to complement this automated validation approach, we also manually inspected modification request descriptions and change logs in the version history to check whether programmers fixed, or at least plan to fix, these reported violations through redesign or refactoring activities. For the rest of the reported violations, we studied the corresponding source code to see whether they include any symptoms of poor design.

7.4.3 Results

We analyzed our results by answering the questions proposed at the beginning of the section.

Q1. Accuracy of Identified Design Violations

Table 7.2 shows the total number of violations identified by our approach $|V|$, the total number of violations that match with automatically reconstructed refactorings $|V \cap R|$, the total number of remaining violations that were confirmed based on manual inspection $|V \cap M|$, the total number of confirmed violations $|CV|$ (which is $|V \cap R| + |V \cap M|$), and the precision, which is defined as the number of confirmed violations out of the total number of reported violations: $\frac{|CV|}{|V|}$.

Table 7.2: Modularity Violations that Occurred at Least Twice in the Last Five Releases

|              | $|V|$ | $|V \cap R|$ | $|V \cap M|$ | $|CV|$ | Pr.  |
|--------------|------|--------------|--------------|-------|------|
| Eclipse JDT  | 399  | 55           | 104          | 161   | 40%  |
| Hadoop Common| 231  | 81           | 71           | 152   | 66%  |

Our approach identified 231 violations that occur at least twice in a five release period of Hadoop, out of which 152 (66%) were confirmed. 81 of them were automatically confirmed and 71 were manually confirmed. Figure 7.4 shows the precision for those violations that occur at least twice and the violations that occur at least three times. With at least three occurrences, we obtain a similar precision of 67% but fewer reported violations. For Eclipse JDT, our approach identified 399 violations, of which 161 were conservatively confirmed (40%). Requiring violations to occur at least
three times increased the precision to 42%. We only discuss the results of requiring at least two occurrences for the rest of the paper because the results of higher occurrence rates are its subsets.

By comparing the results of Hadoop and Eclipse JDT, we first observe that Eclipse is better modularized and more stable: although Eclipse JDT is about ten times larger than Hadoop, less than three times more refactorings were discovered from Eclipse JDT than from Hadoop, showing that it has been less volatile. This is consistent with the fact that only 12% of all the 3767 Eclipse MRs were detected to have violations (in Hadoop, the number is 47% out of the 490 MRs), showing that the changes to Eclipse JDT matches its modular structure better. Because Eclipse JDT is much larger and the violations found are much sparser, it was much harder for us to determine if a violation indicates a problem, hence leading to a lower precision.

![Figure 7.4: Modularity Violation Detection Precision: Hadoop](image)

**In-depth Case Study: Hadoop**

Now we present an in-depth study of Hadoop to demonstrate examples of violations that are (1) automatically confirmed violations, (2) manually confirmed violations, (3) false positives (violations that are not confirmed), and (4) false negatives (refactorings that are not identified as violations).

**Automatically confirmed violations.** In release 0.3.0, our approach found a violation involving `FSDirectory` and `FSNamesystem`. `FSNamesystem` depends on `FSDirectory.isValidBlock` method, but it often changes with `FSNamesystem`. An API-level refactoring was identified in release 0.13.0,
showing that the `isValidBlock` method was moved from `FSDirectory` to `FSNamesystem`. Upon further investigation, we saw that, in the subsequent release, the method was made `private`. In this case, Clio identified this violation 11 releases prior to the actual refactoring.

**Manually confirmed violations.** Our approach identified a violation in release 0.2.0 involving `TaskTracker`, `TaskInProgress`, `JobTracker`, `JobInProgress`, and `MapOutputFile` that does not match with automatically reconstructed refactorings. We searched Hadoop’s MRs and found an open request `MAPREDUCE-278`, entitled “Proposal for redesign /refactoring of the `JobTracker` and `TaskTracker`”. The MR states that these classes are “hard to maintain, brittle, and merits some rework.” The MR also mentions that the poor design of these components have caused various defects in the system.

**False Positive Violations.** Violations in this category cannot be confirmed either automatically or manually. In most cases, we cannot determine if there is a problem because we are not domain experts. As an example, in release 0.4.0, we found a violation containing `ClientProtocol`, `NameNode`, `FSNamesystem`, and `DataNode`. `ClientProtocol` contains a public field with the protocol version number and whenever the protocol changes, this number needs to change. Since `NameNode`, `DataNode`, and `FSNamesystem` implement the protocol, changes to them induce a change to `ClientProtocol`. Although there may actually be a design problem, we are not able confirm it.

**False Negative Violations.** Some reconstructed refactorings are not matched to any violations identified by our approach. There are many micro-refactorings that happen within a class and do not influence the macro structure of the system. Refactorings can also happen for other purposes. Another reason is that some discrepancies only occur once, so our approach cannot tell if they are accidentally changed together or there is a problem, but the developers may have realized and fixed it before it happens again. For example, in version 0.15.1, the `INode` inner class of `FSDirectory` was refactored and extracted into a separate class, and two of its sub-types `INodeFile` and `INodeDirectory` were created so that the `DFSFileInfo` and `BlocksMap` classes can be separated and use specific `INode` subtypes. Our approach did not identify a violation between these classes because they were only involved in a single MR during the time frame we examined.
Q2. Timing of Violation Detection

In Hadoop and Eclipse JDT, our approach identifies a violation, on average, 6 and 5 releases respectively, prior to the releases where the classes involved in the violation were actually refactored or recognized by developers. Figure 7.5 shows the distribution of the confirmed violations over Hadoop releases. Each point in the plot represents a set of confirmed violations, such that the horizontal axis shows the version that the violations were first identified by our approach and the vertical axis shows the version that the violations were refactored or recognized by the developers. Points above 20 in the vertical axis signify that the violations have been recognized by developers but not refactored yet. Most of the points in Figure 7.5 are above the line, indicating that our approach can identify design violations early in the development process so that the designers can be warned to avoid these problems accumulating into severe decay.

Figure 7.5: Timing of Violation Detection (Hadoop)

Q3. Characteristics of Identified Violations.

We further analyzed the symptoms of design problems associated with the detected violations and categorized them into the following four types: (1) cyclic dependency, (2) code clone, (3) poor inheritance hierarchy, and (4) unnamed coupling. The first three symptoms are both well defined and can be detected using existing tools. We call the fourth category unnamed because they are not
easily detectable using existing techniques, to the best of our knowledge. Table 7.3 shows the number of confirmed violations under each category in Hadoop and Eclipse JDT. The cyclic dependency, code clone, and unnamed coupling violations reported in the table are mutually exclusive from each other. The symptoms of poor inheritance hierarchy often overlap with cyclic dependency or unnamed coupling. Next, we provide examples from each category.

Table 7.3: Characteristics of the Violations

<table>
<thead>
<tr>
<th>Subject</th>
<th>Cyclic</th>
<th>Clone</th>
<th>Inheritance</th>
<th>Unnamed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse JDT</td>
<td>72</td>
<td>52</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>Hadoop Common</td>
<td>58</td>
<td>18</td>
<td>37</td>
<td>66</td>
</tr>
</tbody>
</table>

**Cyclic Dependency.** Both systems contain considerable number of cyclic dependencies. For example, in Eclipse JDT, we found that the `JavaBuilder` and `AbstractImageBuilder` often change together, and the code shows that `JavaBuilder` contains a subclass of `AbstractImageBuilder`, and `AbstractImageBuilder` contains a `JavaBuilder`. In a Lattix DSM, there are no symmetric marks to alarm the designer of this indirect cyclical dependency. Similarly, we found that all of the following five files, or their subsets often change together: `JavaProject`, `DeltaProcessor`, `JavaModelManager`, `JavaModel`, and `JavaCore`. It turns out that these five classes form a strongly connected components if represented as a syntactic dependency graph.

**Code Clone.** Some modularity violations detected by our approach involve code clones. In Hadoop version 0.12.0, the detected modularity violation involves the classes `Task`, `MapTask`, and `ReduceTask`. Our approach identified two violations: one involving `MapTask` and `Task`, and the other involving `ReduceTask` and `Task`. Various cloned methods and inner classes from `ReduceTask` and `MapTask` were pulled up to the parent `Task` class in versions 0.13.0, 0.14.0, and 0.18.0. In Eclipse JDT and Hadoop, there are 52 and 18 violations respectively that exhibit symptoms of code cloning. Using clone detectors, it is possible that a much larger number of code clones can be detected, but it may be too costly and unnecessary to refactor all of them. Our approach picks up the ones that happen most recently and most frequently, and provides more targeted candidates to be refactored.

**Poor Inheritance Hierarchy.** The poor hierarchy violations we identified all have the symptoms that the subclasses cause the base class and/or other subclasses to change for different reasons. For example, we identified, in version 0.2.0 of Hadoop, a violation involving the `DistributedFileSystem` and `FileSystem` classes, which was refactored in version 0.12.0: several methods in `DistributedFi-
leSystem were pulled up to its parent, FileSystem, making them available to the other FileSystem subtypes. Another reason is that the subclasses extensively use some methods in their parent class and a push-down method refactoring should have been applied [88]. For example, in Hadoop version 0.14.0, the getHints method was pushed down from the ClientProtocol to its subclass, DFSClient, because it was the only client of this method. They were detected as a modularity violation in version 0.2.0.

In some cases, the parent classes depend on the subclasses and form a cyclic dependency. In Hadoop version 0.1, modification request #51 describes changing the DistributedFileSystem class but its parent class FileSystem and another child of the FileSystem, LocalFileSystem, are also part of its solution. There are no syntactic dependencies between the two sibling classes. By release 0.3, our approach finds that this modularity violation was observed more than three times already. The code shows that the parent FileSystem class contains methods to construct both of the two subclasses. The parent class is thus very unstable because changes to a child require changes to itself and its other children. Our intuition that this is a problematic issue was confirmed when we looked forward through the revision history and found that by release 0.19, the method to construct DistributedFileSystem had been deprecated in FileSystem, in favor of a method in an external class. As a similar example in Eclipse, Scope is the parent of ClassScope and BlockScope, but it constructs both of its children. We categorized these modularity violation as both poor inheritance and cyclic dependency.

**Unnamed Coupling.** The files involved in violations of this category often change together, but they either do not explicitly depend on each other (and are not code clones), or have asymmetric dependency. For example, in Hadoop, DatanodeInfo and DataNodeReport were involved in a violation, which was later refactored. In the modification request comments, the developer says that these classes “seem to be similar” and need to be refactored.

The FSDirectory and FSNamesystem we mentioned earlier is also an example of unnamed coupling. We detect this violation because the only allowed change order is from the interface of FSDirectory to FSNamesystem. But the revision history shows that changes to FSNamesystem often cause FSDirectory to change. In the corresponding syntactical DSM, these two classes reside in the same package, and FSNamesystem depends on FSDirectory. Using a Lattix DSM, the user can mark that FSDirectory should not depend on FSNamesystem so that if FSDirectory explicitly refers to FSNamesystem, Lattix will raise a warning. However, in reality, FSDirectory never explicitly refers
to FSNamesystem, although it often changes with FSNamesystem. Table 7.3 shows that in Hadoop 66 out of 152 of the confirmed violations fall into this category (In Eclipse, the number is 25 out of 161).

We are not aware of existing techniques that detect these violations that do not fit to pre-defined symptoms of poor design.

### 7.5 Discussion

The quality of our modularity violation detection approach depends heavily on the availability of modifications requests and their solutions. For small-scale projects or projects without version control systems, it is hard to apply our approach.

When calculating evolutionary coupling, how long a version history is enough? The answer depends on the specific project and how to determine the best threshold is our ongoing work. In the evaluation, we use all available revision histories to determine change couplings. To address this issue, as a future work, we plan to apply our stochastic dependencies for performing history-based impact analysis. Additionally, changing the number of versions used for change coupling may change the results. Our decision of only considering the five most recent releases in evaluation when determining violations is based on the fact that the results do not greatly differ when we consider more versions. Again, this heuristic may vary with different projects.

The selection of a starting impact set can significantly affect the accuracy of violation detection. We use the most highly coupled elements in a MR solution as the starting impact set. However, other heuristics can be used for selecting a starting impact set. Identifying such heuristics and seeing how they affect the accuracy of violation detection is an ongoing future work. Automatically recovering the original starting impact set of an MR is an active, but immature, area of research (e.g. Kagdi et al. [123], Antoniol et al. [12]). Such techniques try to reconstruct what developers would have first modified in fulfilling an MR. As these techniques mature, we can evaluate their effectiveness in improving our approach’s accuracy.

Since we only applied our approach to two subject systems, we cannot conclude that its effectiveness generalizes to all software; however, we did choose projects of different characteristics to begin addressing this issue. Additionally, we cannot guarantee that the MRs used in the evaluation are not biased. As Bird et al. [19, 29] showed, MRs that have associated change sets may not be representative of all the MRs. For example, although we claim to identify violations for actively-developed parts of a system, the collected MRs may not include the most active parts of the system.
We use a frequency heuristic to prioritize the identified modularity violations (i.e., violations that repeatedly occur in maintenance tasks are given highest priority). Exploring other heuristics to prioritize modularity violations is a future work. For example, certain types of violations may be easier to resolve than other and developers may want to address those violations first. With information on ease of fixing violations, we can start to build a return-on-investment (ROI) model that prioritizes violations based on an expected benefit measure. Building such prioritization models remains a future work.

Some violations detected by our approach may not embody any design problems but reveal valid semantic dependency, as shown in previous work [202, 204]. But our experiments show that considerable number of violations indeed reflect design problems. The accuracy of our approach also depends on how accurate the ACN model embodies design decisions and their assumption relations. The ACN model we used in this paper were automatically generated from UML class diagrams derived from source code. Some dependencies can only be reflected in other design models, such as an architectural description. It is possible that these dependencies are missing from the ACN model, hence causing false positives. The violation we discussed in the previous section that contains ClientProtocol, NameNode, FSNamesystem, and DataNode is such an example. A future work is to improve the accuracy of our approach by using high-level architectural models in addition to reverse-engineered source models.

7.6 Conclusion

Parnas’s original definition of a module means an independent task assignment, and his information hiding principle advocates separating internal design decisions using an interface to allow for independent evolution of other modules. Though this definition of modularity is inherently inseparable from the notion of independent module evolution, existing approaches do not detect modularity violations by comparing how components should change together and how the components actually change together.

This chapter presented a novel approach of identifying eroding design structure by computing the discrepancies between modularity-based impact analysis and history-based impact analysis. We evaluated our approach using the version histories of Hadoop Common and Eclipse JDT. We conservatively confirmed hundreds of reported violations to be correct. The result also shows that detected modularity violations exhibit various symptoms of poor design, showing advantages of our approach.
in contrast to bad code smell detection techniques that find only pre-defined set of poor design symptoms, without regard to the system’s original design structure nor its evolution history.
8. Software Tools Support

In this chapter, we describe the tools support for the approaches presented in previous chapters. The software we contribute with this thesis can be separated into three categories: shared libraries, model conversion tools, and design analysis tools. Both the model conversion tools and the analysis tools use the shared libraries to simplify their implementation to reuse common code. The model conversion tools, as shown in Figure 8.1, enable ACN analyses to be performed on existing design models by converting the existing models to ACN models. The design analyses tools, as shown in Figure 8.2, implement the various modularity analyses described in this thesis. In the following sections, we describe the software from each of these categories.

![Diagram showing tools support infrastructure: Model Conversion](image)

**Figure 8.1: Tools Support Infrastructure: Model Conversion**

8.1 Shared Libraries

To support the set of design analysis tools contributed by this thesis, we provide two supporting libraries: Tellus and Caelus.\(^1\) These libraries provide the shared data structures and utilities that serve as a foundation for the tools. In essence, these libraries realize the most important design rules to the software infrastructure—enabling parallel develop of many of the tools. As these are the critical design rules of software framework, changes to them can have drastic ripple effects throughout the system.

---

\(^1\)In Roman mythology, Tellus is the personification of the Earth and Caelus is the personification of the sky. To our software infrastructure, the **Tellus** library provides the groundwork for ACN analysis and low-level (i.e., console-based) applications. The **Caelus** library provides the support for graphical interfaces to be built on top of the low-level applications.
8.1.1 Tellus

The Tellus library provides the fundamental data structures and utilities for ACN analysis. An augmented constraint network is represented in the Tellus library via the Context class. This class contains:

- a name, for ease of identification.
- the constraint network, as encapsulated by the ConstraintNetwork class. The ConstraintNetwork itself contains the set of variable names and their domain values, and a collection of abstract syntax trees (ASTs) modeling the constraints.
- the dominance relation, as represented by a digraph: Digraph<String>. The vertices of the digraph are labeled with the variables of the constraint network, and an edge \((u, v)\) models that variable \(v\) dominates variable \(u\) (i.e., \(v\) is a design rule of \(u\) so \(u\) cannot influence \(v\)).
- the pairwise dependency relation, as represented by a digraph: Digraph<String>. The edges of the digraph show the pairwise dependence of variables from the constraint network. In essence, this structure is the graph representation of the design structure matrix.
- the design automaton, as encapsulated by the Automaton<Solution, Assign> template class. In addition to the automaton itself, the Context also separately maintains a list of the solutions to the constraint network (i.e., states of the automaton).
• the clustering, as modeled by the `Clustering` class hierarchy. The `Clustering` class and its two subclasses (`ClusterSet` and `ClusterItem`) follow a composite design pattern [92] to model the tree structure of a clustering. The `ClusterItem` class represents a leaf of the tree and encapsulates a single variable from the constraint network. The `ClusterSet` class represents an inner node in the tree and aggregates a (possibly empty) sequence of `Clustering` objects. Together, these classes model a nested ordering of variables from the constraint network and allow us to visualize the DSM (e.g. with a design rule hierarchy view).

We emphasize that the `Context` class is simply a container for these components and their construction is handled by other parts of the software. Because of this separation of construction from storage, certain elements of a `Context` may not always be populated, which is desirable from a performance perspective (e.g. design automaton is not needed for processing BACN models).

The `Project` class encapsulates a set of ACNs (i.e., `Contexts`), which have been decomposed using Cai and Sullivan’s DECOMPOSE-MODULES algorithm [50]. A `Project` consists of the original ACN (called the primary context) and the set of decomposed ACNs (called the auxiliary contexts). As with the `Context` class, the `Project` class is simply a container for the ACNs and does not provide an implementation of the decomposition algorithm.

In addition to algorithms to process and manipulate the components of the ACN (e.g. graph traversal algorithms), the Tellus library also provides other data structures and algorithms that are not directly related to ACN analysis. For example, the `Hasher` class provides convenience methods for computing hash keys. The `OptionsParser` and related classes simplify the processing of command-line arguments.

### 8.1.2 Caelus

The Caelus library provides simple utilities developing graphical user interfaces (GUIs) on the Java Swing framework. In particular, since most of the tools in our infrastructure have both console-based and GUI versions, the Caelus library provides support for displaying the console output in a simple text box (via the `TextAreaOutputStream` class).

### 8.2 Model Conversion Tools

To aid the adoption of our design analysis techniques in software engineering practice, we developed a framework for converting existing design models into ACN models. Figure 8.1 depicts these
tools and their relationship in a layered architectural style [174]. From the diagram, we see that the model conversion tools all rely on an additional shared library PORTUNES. The tools in this infrastructure can be categorized into two types: design model converters and code reverse engineering tools. We describe each of these categories separately below.

8.2.1 Design Model Converters

The design model converters in our software infrastructure, transform design models (e.g. UML class diagrams) into ACN models. JANUS\textsuperscript{2} is the primary tool in this category, using plugins to handle specific input file formats. The PORTUNES\textsuperscript{3} library defines the design rules for creating plugins to JANUS. Hence, the PORTUNES library’s design rules allow the independent and parallel development of converters for various design models. In the following subsections, we further detail PORTUNES, JANUS, and the implemented plugins.

PORTUNES

PORTUNES is both the name of our shared library and the model that JANUS uses. The PORTUNES library defines the model specification (via an XML Schema file) and provides utilities, such as checking for conformance to the model specification. The PORTUNES model is an XML-based representation that captures binary relations between software elements.

JANUS

JANUS consists of a two stage conversion process: a format-dependent stage and a format-independent stage. In the first (format-dependent) stage, JANUS selects an appropriate plugin (Camenae) for the input file format. It invokes the plugin to convert the input file to the PORTUNES format, extracting the relevant dependency information. Plugins are configured dynamically at runtime so that the introduction of new input file formats do not require recompiling JANUS. We use a configuration file that maps file name extensions (denoting file format) to the appropriate plugin to use.

With a single model format (PORTUNES) produced by the first stage of JANUS, the second (format-independent stage) stage uniformly applies formalization heuristics to convert the design

\textsuperscript{2}Janus is the Roman god of doorways and beginnings. In our software infrastructure, the JANUS tool provides the “doorway” for existing models to ACN analyses.

\textsuperscript{3}Porunes is the Roman god of keys and doors. Our PORTUNES library provides the “key” specifications to interfacing with the JANUS tool.
model into an ACN model. Janus currently contains heuristics for formalizing UML class and component diagrams. The heuristics for class diagrams were presented in Chapter 4. The heuristics that Janus uses for formalizing component diagrams were contributed by Sethi [172]. The first stage of processing is skipped if Janus is directly given a Portunes model as input (e.g. with Ursatz).

Camenae

The plugins to Janus are called Camenae. Each Camena handles a specific input file format and is mapped to a unique set of file name extensions in the Janus configuration. Our current implementation includes support for the following input file formats:

- GNOME Dia\(^5\) (with the file name extension dia) is an open-source drawing tool that supports creating UML models. Dia actually exports two variants of file: a plain-text XML format and a compressed version of the XML format. While both variants use the same file extension, our Camena determines the specific variant when processing the input file.

- IBM Rational Rose\(^6\) (with the file name extension ptl or mdl) is a UML drawing tool that supports various diagram types including class, component, and sequence. Our current Camena implementation only handles the class and component diagrams in Rational Rose files.

- Moka is our tool (described below, with the file name extension moka or mokaz) for reverse engineering UML class diagrams from Java software.

- XML Metadata Interchange (XMI) \(^7\) (with the file name extension xmi) is a standard, XML-based file format for representing UML models. Various UML tools (e.g. Rational Rose, Enterprise Architect\(^7\)) support exporting UML models in XMI format.

8.2.2 Reverse Engineering Tools

Our reverse engineering tools recover design models (e.g., class diagrams) from code (source or binary) and the design models can then be converted to ACN models using the tools described in the previous subsection. These tools can generate Portunes models directly (e.g. Ursatz), or generate a separate design model (e.g. Moka) and use a Camena for conversion to the Portunes format. The two currently implemented reverse engineering tools we have are Moka and Ursatz.

\(^4\)In Roman mythology, the Camenae were goddesses of childbirth and fountains. The Camenae (singular: Camena) in our infrastructure deliver the birth of Portunes models from existing models.

\(^5\)http://live.gnome.org/Dia/

\(^6\)http://www-306.ibm.com/software/awdtools/developer/rose/

Moka

The Moka\textsuperscript{8} tool analyzes Java bytecode and exports a UML class diagram (in an XML-based Moka file format). We also provide a tool to convert Moka files into GraphViz DOT files for rendering the class diagrams. Our tool exports a Moka file rather than a Portunes file in order to retain additional information from the class diagram that is not used by the Portunes model (e.g., number of method classes, rather than just the existence of a method call). This additional information is useful for some of our other analyses, such as computing static coupling measures \cite{39} for impact analysis.

Ursatz

Ursatz\textsuperscript{9} is a tool that exports Portunes files from analyzing .NET assemblies. This tool can be used as either an add-in to Visual Studio or as an external application. Due to the benefits of using for Moka file format for other analyses, in the future, we plan to export Moka files from Ursatz rather than directly to the Portunes model.

8.3 Design Analysis Tools

Figure 8.2 shows the layered architecture style of our design analysis tools. Again, all components of our framework rely on the Tellus library as the primary design rules. This infrastructure consists of two suites of tools: Parcae and Clio. The Parcae suite of tools is for ACN analyses while the Clio suite of tools enables history-based analyses of software repositories. We describe both Parcae and Clio in the following subsections.

8.3.1 Parcae

The Parcae\textsuperscript{10} suite of tools consists of the Parcae console-based tool, the Nona library for plugin development, the plugins themselves (called Civitates), and the Titan DSM viewer. Similar to Janus, Parcae dynamically configures the Civitates at runtime to avoid the need to recompile when introducing new plugins. Parcae uses an XML-based configuration file that defines which plugins to run, in what order, and where to find plugin libraries for dynamic loading.

\textsuperscript{8}The moka pot is an espresso coffee maker.
\textsuperscript{9}In Schenkerian musical analysis, the Ursatz is the most basic and fundamental structure of harmony. We use the name Ursatz as a reference to the C# programming language, which was the initial target for our tool.
\textsuperscript{10}The Parcae are Roman deities that control the birth and death of all people. As one of the Parcae, Nona is the goddess of pregnancy that spins the thread of life. In our context, the Nona library provides the constructs to enable the “life” of plugins that run in Parcae. The Parcae application, hence, controls the lifecycle of the plugins. Keeping with the analogy, the plugins to the Parcae are Civitates (Roman citizens).
The internal architecture of Parcae is a combination of batch and repository styles [174]. Plugins are sequentially executed in a batch style and the plugins communicate using a central repository. For its central repository, Parcae extends the Project concept from the Tellus library in the form of the Environment class. Extending Project from the Tellus library makes the ACN the primary shared context in the repository.

Nona

The NONA library defines the design rules for creating a plugin to Parcae. Specifically, this library defines the interface that all plugins must realize. It includes an abstract implementation of this interface that provides utilities for implementing plugins (e.g., loading of plugin description and command-line usage information from a configuration file). The NONA library is also where the central repository is defined.

Civitates

A plugin to Parcae is called a Civitas (plural: Civitates). Figure 8.2 separates the Civitates into two sets: those that are based on our Minos tool and those that are not. Below, we detail both types of Civitates currently available in our infrastructure.

Minos Plugins. Minos is our reimplementation of Cai and Sullivan’s Simon [49] tool for DSM derivation. We implement a set of Civitates on top of Minos (shown as the left group in the Civitates box of Figure 8.2). We describe each of these Civitates below.

- The acn-io plugin allows (1) the parsing of an ACN file and constructing of a Context object in the central repository, and (2) the writing of a Context object to a file.
- The clsx-io plugin allows (1) the parsing of a cluster file (with file name extension clsx) and constructing the Clustering objects in the central repository, and (2) the writing of a Clustering object to a file.
- The dsm-io plugin allows (1) the parsing of a DSM file and constructing of a Digraph, and (2) the writing of a dependency Digraph object to a file.
- The decomposer plugin implements the Cai and Sullivan’s [50] Decompose-Modules algorithm for splitting an ACN. This plugin also implements their algorithm for combining the design automata (DAs) of the sub-ACNs back into a single DA.

11In Greek mythology, Minos was a king of Crete. As our Minos tool is a reimplementation of Cai and Sullivan’s Simon [49] tool, Minos is also an anagram of Simon.
• The cia plugin implements Cai and Sullivan’s algorithm [49, 50] for performing change impact analysis (CIA) on the an ACN.

• The solver plugin leverages the Kodkod [188] tool to enumerate the solutions to the constraint network and store the solutions in the central repository.

• The pwdr plugin uses the solutions to the constraint network to construct the design automaton (DA), populating the transitions in the DA to obey the dominance relation.

**BACN.** The BACN plugin implements the two BACN-related algorithms detailed in Chapter 4: computing pairwise dependency relations (PWDRs) and performing change impact analysis (CIA).

**DR Hierarchy.** The dr-hier plugin implements the DECOMPOSE-HIER algorithm described in Chapter 5 to derive a design rule hierarchy (DRH) from an ACN model. The resulting hierarchy is placed back into the central repository as a clustering.

**DR Impact Analysis.** Chapter 7 introduced the dr-predict plugin that performs impact analysis (IA) by applying Robillard’s [163] algorithm on a DRH digraph. This plugin is implemented as a Civitas for Parcae and generates an impact scope.

**Spreadsheet.** We provide a spreadsheet plugin that enables the exporting of DSMs to the Microsoft Excel file format. This Civitas uses the clustering specified in the central repository to order the rows and columns of the DSM when exporting.

**Debug.** To assist in the debugging and testing Parcae and other plugins, we also provide the pause, timer, and size plugins. The pause plugin causes Parcae to wait until user input is acquired. The timer plugin reports the amount of time elapsed since it was last executed, allowing users to measure the performance of other plugins. The size plugin reports the number of variables, constraints, and dominance relation pairs in the ACNs in the central repository.

**Titan**

Our Titan\textsuperscript{12} tool is a design structure matrix (DSM) viewer, enabling the visualization of DSMs exported by Parcae. Using cluster files (either exported by Parcae, some other tool, or manually defined) the rows and columns of the DSM can be ordered for different perspectives on the design.

\textsuperscript{12}The Titans were a race of immortals in Greek mythology.
structure. TITAN allows users to modify the clustering and, using certain Civitates, export DSMs to spreadsheet files.

### 8.3.2 Clio

Clio\(^\text{13}\) is our tool for performing analyses that require historical data. The architecture of Clio consists of a two level hierarchy of plugins. The top-level plugins, called Poleis\(^\text{14}\) merely provide shared functionality to a second level of plugins. We currently provide two Poleis with Clio: EXTRAPOLIS for extracting information from data sources (e.g. revision history archives, mailing list archives) and ANAPOLIS for analyzing the revision history data. The reasoning for this two level hierarchy is that we can categorize plugins into a small number of categories (i.e., data extraction and data analysis) and the plugins within the same category often share common code.

Similar to PARCAE, Clio also employs a repository style architecture; however in the case of Clio, the repository is backed by an external database. The PHANES\(^\text{15}\) library defines the design rules in the Clio suite of tools, including the interfaces for the Poleis implementation and common repository concepts (e.g. transactions in revision history). By using an external database for the repository, Clio allows the plugins easily to extend types (i.e., tables) for data they use.

Next, we describe the plugins available for Clio.

**Extrapolis.** EXTRAPOLIS plugins extract information from an external data source (e.g. file) and store the data in the Clio database for use by other plugins. We currently have two EXTRAPOLIS plugins implemented: one for extracting SVN revision history data, and one for extracting communication data from mailing lists of Apache projects.

**Anapolis.** ANAPOLIS plugins analyze the historical data stored in the Clio database. Below, we describe each available ANAPOLIS plugin.

- The *logical* plugin analyzes the revision history to extract association rules, storing the association rules with their support and confidence values back into the database. (The *extrapolis* plugin referred to in Chapter 7 was actually a combination of EXTRAPOLIS and the *logical* plugin. For simpler illustration, we combined these two plugins into one in that discussion.)

---

\(^{13}\)Clio is the Greek muse of history. In our context, the Clio tool serves to extract and analyze historical data (e.g. revision history).

\(^{14}\)From the ancient Greeks, a polis (plural: poleis) is a city. The suffix *-polis* is often found in city names. Hence, we use that suffix in naming our plugins also.

\(^{15}\)In Greek mythology, the Phanes is a primeval deity that generates new life.
• The *evol-predict* plugin uses the association rules extracted by the *logical* plugin to perform change impact analysis, given a specified starting impact set. When detecting modularity violations, the starting impact set is automatically derived from the change set.

• The *detect* plugin implements the violation detection algorithm described in Chapter 7 (i.e., comparing the modularity-based and history-based impact scopes) and reports the discrepancies found.

• The *stochastic* plugin performs impact analysis using stochastic dependencies. This plugin provides implementations for all four $\lambda$ sequences presented in Chapter 6, and allows for easy additional of new $\lambda$ sequences.
9. Conclusions

In this thesis, we contributed a framework that has the potential to help developers maximize concurrent work during software development and maintenance. Specifically, we presented: (1) the binary augmented constraint network model to support the contributed approaches and a set of heuristics for converting UML class diagrams into our formal model, (2) the design rule hierarchy model for identifying independent tasks from a design and assessing how well a design supports concurrent work, (3) the Clio framework to identify modularity violations and prioritize their severities, and (4) the stochastic dependency theory, generalizing evolutionary dependencies to include a temporal dimension.

We intend to continue researching modularity analysis techniques that facilitate task parallelism in software development. Our future work includes evaluating our framework with large-scale software projects and in live development environments. As discussed in the various chapters of this thesis, we plan to extend each of the contributed models and theories as future work. For example, the design rule hierarchy can be extended to include quantitative measures on the independence of modules/tasks. Our stochastic dependency theory can be applied to map software components to their owners, in order to estimate coordination needs. Stochastic dependencies can also be used for identifying modularity violations. Improving the prioritization of modularity violations based on a quantitative return-on-investment model is also a possible future direction of research. The possible directions for future research based on this thesis are extensive.
Bibliography


[92] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, November 1994.


[101] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 1–12, May 2000.


Vita

Sunny Wong (formerly Sunny Huynh) received a Bachelor of Science in Computer Science from Drexel University in 2007, graduating summa cum laude. In 2008, he was the recipient of the College of Engineering Outstanding Graduate Student Research Award. That same year, Sunny received the Best Student Poster Award at the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) for his work, entitled Design Rule Hierarchy and Model Transformations.

Sunny is a member of the Association for Computing Machinery (ACM) and the Institute for Electrical and Electronics Engineer (IEEE) Computer Society. He is a member of Upsilon Pi Epsilon (UPE), the international honor society for computing and information disciplines, and served as president of Drexel University’s student chapter from 2007 to 2010. For his leadership in UPE, Sunny received the 2010 UPE Special Recognition Award.

Book Chapters


Conference Papers


