Abstraction Patterns for Reverse Engineering

by

Robert Ian Bull

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada 2002

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Ian Bull

I authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Ian Bull

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Acknowledgements

First and foremost I would like to thank my supervisor Andrew Malton, not only for his support and encouragement, but for putting up with my unscheduled meetings and actually taking the time to help me realize my intuition. Your help was invaluable.

I would also like to thank my readers Ric Holt and Charlie Clark. You both influenced me throughout my career at Waterloo, it is only fitting that you should be readers for my thesis.

I can't forget the support my family gave me, both financially and emotionally throughout my entire academic career. I would have never even considered taking on this task if it wasn't for your help and encouragement. Mom, Dad, Joanna and Jacquie, thank-you.

I would also like to say a very special thank-you to Tricia. Your love and companionship was always steady and you ensured that I played as hard as I worked.

Finally I would like to thank the boys, Wayne, Pat, Mike, Jen, Josh, Rutherford, Blex, Woz, and Andrew "0" Trevors. You guys always reassured me that, *"I would be more relaxed tomorrow and get more done, if I hung out with you guys today."*

Abstract

As software systems continue to grow in size, often requiring several millions of lines of code, automated methods to assist a software engineer's understanding are indispensable. Currently, reverse-engineering toolkits exist that extract and present information to a software engineer about a software system. The views these applications provide are produced by deducing new information and removing extraneous information from extracted program artifacts. It is this process which we consider to be abstraction.

In order to handle large programs, the programs must be split into smaller, manageable pieces. Abstractions can then be performed on these smaller pieces and the abstracted parts linked together.

In this thesis we formalize the notion of abstraction and linking to show how these two ideas can be used together to produce a number of useful views of large software systems. We have also cataloged the ways in which abstractions are performed in practice. Four reverse-engineering tools were studied and from these tools, eleven patterns of abstraction were uncovered. These patterns can be used as a basis for designing or enhancing reverse-engineering toolkits.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

All truth passes through three stages. First, it is ridiculed.
Second, it is violently opposed. Third, it is accepted as
being self-evident.
– Arthur Schopenhauer (1788-1860)

TECHNOLOGY shapes the world we live in, and software drives this technology. Software has become increasing expensive to develop properly. Many years of design, implementation, and testing have gone into the systems we use on a daily basis such as: bank machines, medical software and a large assortment of embedded systems. Often these systems are built up over generations of developers, throughout many changes in hardware, changes in management and corporate restructuring. As maintenance activities are performed on these legacy systems, it is important for the developer to attain a high

level of understanding of the system, so that when a change is made, he can determine what parts of the system will be affected.

1.1 Motivation: The Problem of Scale

Pine® – a **P**rogram for **I**nternet **N**ews & **E**mail – is a tool for reading, sending, and managing electronic messages [oW]. Developed by the Computing and Communications department at the University of Washington, Pine is freely available for non-commercial use and can be deployed on several platforms including Microsoft Windows and many variants of Unix. Pine version 4.44 was released in January 2002 and contains 581 source files divided into 338 C files and 243 header files. The entire software system contains 226,832 lines of code[1].

In order to help understand how Pine is structured, a factbase was generated containing all the source code artifacts and relationships between them. The factbase can be queried in order to retrieve static information about the software such as function call information, data references, subsystem structure and type information. The factbase was generated using CPPX [MDH01], an open-source fact extractor for C and C++. The factbase for Pine contains over 1.5 million vertices and over 2.5 million edges. The schema for the factbase was difficult to comprehend, containing 34 vertices and 145 edges. The entity relationship diagram for this schema can be seen in figure 1.1. Because of the complexities in the schema, it is hard to understand how the vertices are related, making queries very difficult to pose.

[1]These data were obtained by counting the number of lines of code in all the .c and .h files after removing comments and white space.

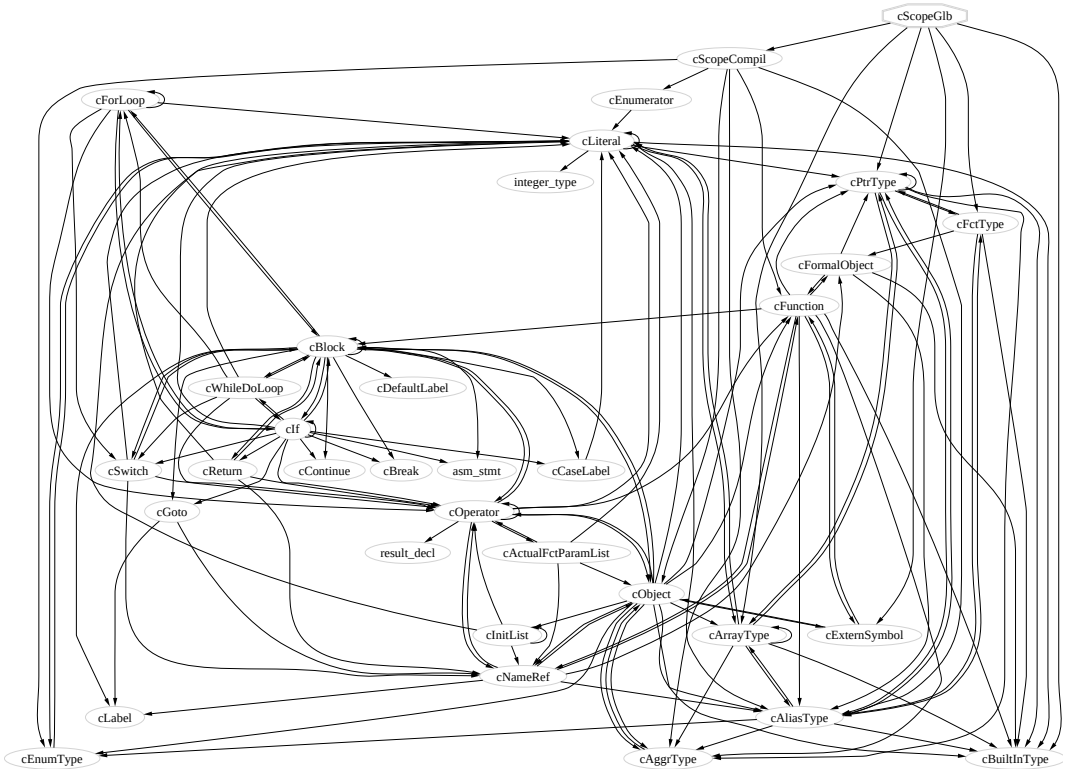

Figure 1.1: Schema for the Pine Factbase

The complexities in models such as the ones described here is the motivation for this thesis. It is hoped that by studying how these models can be abstracted, more concise and easier-to-understand models can be generated while still providing enough information to make the data useful.

1.1.1 Scale Problems

The performance of many reverse-engineering tools deteriorates as the size of the software grows over a few hundred thousand lines of code, however it is systems of this magnitude that cannot be understood without aid. As well, artifact extractors such as CPPX produce large amounts of output for even the smallest programs due to template, macro, and header expansion, standard library inclusion and the intention of these tools to produce output at a high level of detail. It becomes more difficult to navigate the output of a reverse-engineering tool then it does to explore the source code directly.

When converted to C++ and extracted using CPPX, the `"Hello, World!"` program which Kernighan and Ritchie made famous [KR88] produces 4MB of TA [Hol97] and 18MB of GXL [GXL] (Section 2.4.1). However, most of this data is not needed to help a software engineer understand the program. This thesis will look at how the data produced by such extractors can be reduced while still allowing for interesting design-recovery activities.

1.1.2 Limitations of Architectural Diagrams

Logical Architectural Blueprints [Kru95] are an excellent way to get a high-level picture of a large software system. From figure 1.2 [BHB99] it can be seen which subsystems

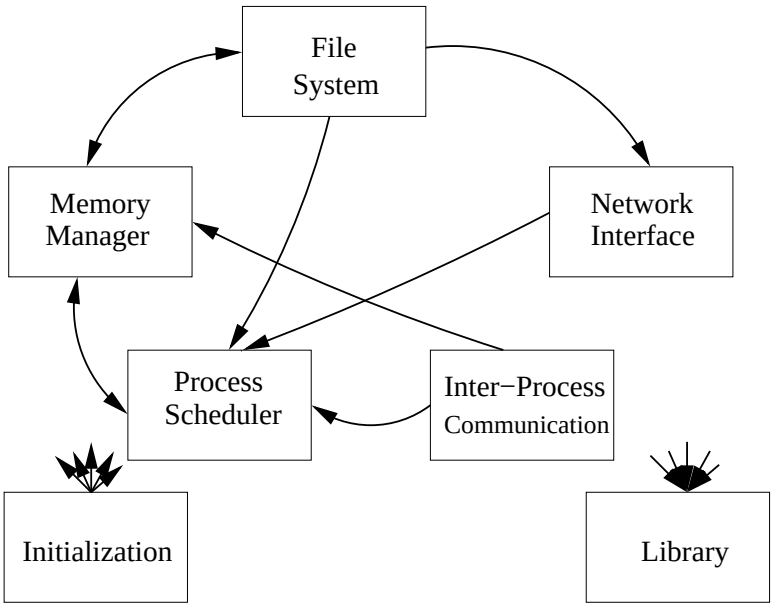

Figure 1.2: Linux Conceptual Architecture

exist in the Linux kernel [LIN] and how these subsystems interact. However, architectural diagrams are usually not sufficient support for maintenance activities because the nodes and edges found in these diagrams do not relate easily to the source code. In the sequel we examine and categorize methods of creating different views from the same data that produced these architectural diagrams, views that more directly relate to the maintained source.

1.2 Overview: An Abstraction Pattern Language

As Perry and Wolf explain [PW92], ”there are a number of different views needed of the software architecture for the various uses and users.” In this thesis we have taken the implementation or builder’s detailed view. This is the same view that Kruchten [Kru95]

has described as the *logical view* and Bowman *et al.* [BHB99] calls the *concrete architecture*. The current approach used by many reverse-engineering tools to uncover the implementation view, is to extract a large factbase of information, abstract the factbase to the architectural level, and then lay out the model diagrammatically as in figure 1.2. If more information is required the abstraction algorithm does not "lift" to such a high level. **It is the goal of this thesis to study the ways in which these abstractions are performed to determine if the types of abstractions can be specified as a set of patterns.**

It is important that the same abstraction be obtained regardless of whether the abstraction is applied to the whole system, or to the system in parts and the parts combined afterwards. This property is known as the commuting property and holds if and only if the abstraction of the factbases $\mathcal{X}$ and $\mathcal{Y}$ is the same as the abstraction of factbase $\mathcal{X} + \mathcal{Y}$, where $\mathcal{X} + \mathcal{Y}$ is the union of factbases $\mathcal{X}$ and $\mathcal{Y}$. By applying the abstraction on the individual compilation units, large programs can be abstracted in pieces and then the individual pieces can be assembled at a higher level where there is less data.

1.3 Related Work

This thesis studies the types of abstractions performed by the reverse engineering community and attempts to categorize these abstractions into a collection of patterns. We have not found any other general survey of the abstractions performed. Although tools such as Bunch [MMCG99] have claimed to use techniques such as Rigi's [RIG] removal of omni-present nodes, there has been no formalization of the term abstraction and a listing of techniques that can be classified as abstractions.

Using a pattern language to describe a collection of related things, has been used by many people in the area of software engineering [BMR+96, GHJV95]. We are aware that the use of a pattern language as a form of classification does not actually provide a concrete solutions for any problem, but we do believe that they are an important aid for allowing people to communicate ideas to one other.

1.4 Organization of Thesis

The body of the thesis is organized as follows. Chapter 2 establishes the context in reverse engineering and program understanding. Chapter 3 explores and defines various notions of abstractions and shows how they are of use in uncovering design. Chapter 4 defines *abstraction pattern* and outlines five patterns which can be applied on a factbase directly. Chapter 5 continues to outline abstraction patterns. The patterns given in this chapter are directed at reducing the complexity of the schema to which the factbase adheres. Chapter 5 also shows the power of composing patterns to provide powerful abstractions. Finally, Chapter 6 summarizes our results and presents future research in the area of Abstraction Patterns.

CHAPTER 2

Background

Each problem that I solved became a rule which served

afterwards to solve other problems.

– Rene Descartes (1596-1650)

In this chapter we present the notion of software reverse engineering, what it is and how it is performed (Section 2.1). The reverse-engineering process is discussed (Section 2.2), along with the current practice for extracting, manipulating and presenting software artifacts. Different methods of representing software systems to help a software engineer better understand the system are introduced (Section 2.3), along with background information regarding directed typed graphs and graph schemas, our choice of model for representing programs. Finally, different schemas used in reverse-engineering tools are presented (Section 2.4).

2.1 Reverse Engineering

Reverse engineering is the process of analyzing a subject systems to *1) identify the system's components and their interrelationships* and *2) create representations of the system in another form or at a higher level of abstraction*. It has been an area of academic research for over 15 years [CI90]. The field evolved as a major part of the software life cycle when legacy applications grew so large that it was impossible, even for the most senior developers, to completely understand the entire system. In recent years, the field expanded to aid open source development [BHB99, GL00, TGLH02], assist new team members "Ramp-Up" on an existing project [SH98] and the field has even started to assist the web development community [HH02]. The field came under media spotlight in the late 1990's with the assistance the research provided in helping with the Year 2000 problem (Y2K) [BERP98, CSDM01]. There is also several conferences dedicated to the study of software reverse engineering including the International Workshop on Program Comprehension [IWP], and the Working Conference on Reverse Engineering [WCR].

As software systems grow in size they become more difficult to understand. The code base is larger, the module interrelationships more complex, and the dependencies cannot be understood by a single developer. Design decisions are lost or obscured. Changes to one part of the source code may have side effects in other parts of the source code that cannot be easily understood. Many reverse-engineering tools [CNR90, CSDM01, CC01, CH95, FHK+97, LA97, MK88] have been presented to both the research and industrial communities to help uncover the software architecture [BHB99], software evolution [TG02, BR02], class diagrams [BI02], semantic information [BTMG02], and other relationships found within a code base. Typically, these tools only target a single, abstract

representation and so, many small details such as expressions and local variables are not recovered. During maintenance however, a software engineer will be required to make changes to the source code in order to *1) adapt the system to a new environment, 2) add new user requirements, 3) fix errors,* and *4) prevent problems in the future* [LS80]. If a reverse-engineering tool does not provide information about the source code, then the tool cannot be easily used for maintenance activities. To insure that a reverse-engineering tool can be used for maintenance activities, the tool should be designed with the requirements of re-engineering in mind. Reverse engineering is only half of the re-engineering process [KWC98]. Reverse-engineering tools should have the ability to integrate with other software tools for tasks such as clone detection [BYM+98, BMD+00], architectural repair [TH99] and forward engineering [CI90].

Chikofsky and Cross's definition of reverse engineering has two distinct parts. The definition includes the identification of the system's components and their interrelationships and the abstract representation of the system. Mendonça and Kramer [MK96] identified these two steps as 1) *the extraction phase*; and 2) *the analysis phase*. Armstrong and Trudeau [AT98] consider reverse engineering to be a three step process by splitting the analysis phase into 2 parts: 1) *classification or fact manipulation*; and 2) *visualization*. The Software Architecture ToolKit (Swagkit) [Sof] has refined the process by splitting these stages into even smaller processes such as Extract, Abstract, Link, Cluster, Layout, and Visualize.

In this thesis, we consider the stages of reverse engineering to include:

- *Extraction phase:* The extraction phase is used to uncover the source artifacts from a factbase and determine the relationships between these artifacts.

- *Abstraction phase:* The abstraction phase manipulates these entities and relationships in order to deduce a form that can be used by other tools or for human analysis.

- *Presentation phase:* The presentation phase puts forth the artifacts in a way that makes them useful for further tasks such as querying or automated re-engineering.

The extraction phase includes the extraction of source artifacts and their relationships and the clustering of entities into higher level subsystems. During artifact extraction, methods are placed in the classes which define them and classes are placed in the files which define them. The process of clustering files into higher level components such as subsystems cannot be extracted from the source code, but we still consider this activity extraction. We consider this step to be part of the extraction phase because this information is usually extracted from sources such as documentation, users mental model, directory structure or a clustering tool such as Bunch [MMCG99] or ACDC [TH00]. This clustering is then fused with the source extraction to derive the entire factbase.

The abstraction phase is the focus of this thesis and is used to create alternative views of the extracted factbase.

The presentation phase can include SQL queries, layout and visualization, natural language queries or any other means by which the data can be exported for future use. It is important to understand that reverse engineering is more than uncovering artifacts about a software system, but extends to the analysis and presentation of these artifacts both for human understanding and automated re-engineering.

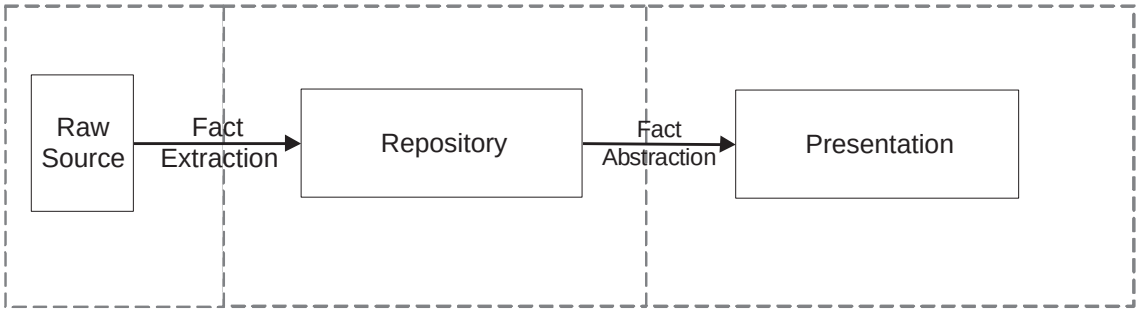

Figure 2.1: General Reverse Engineering Process

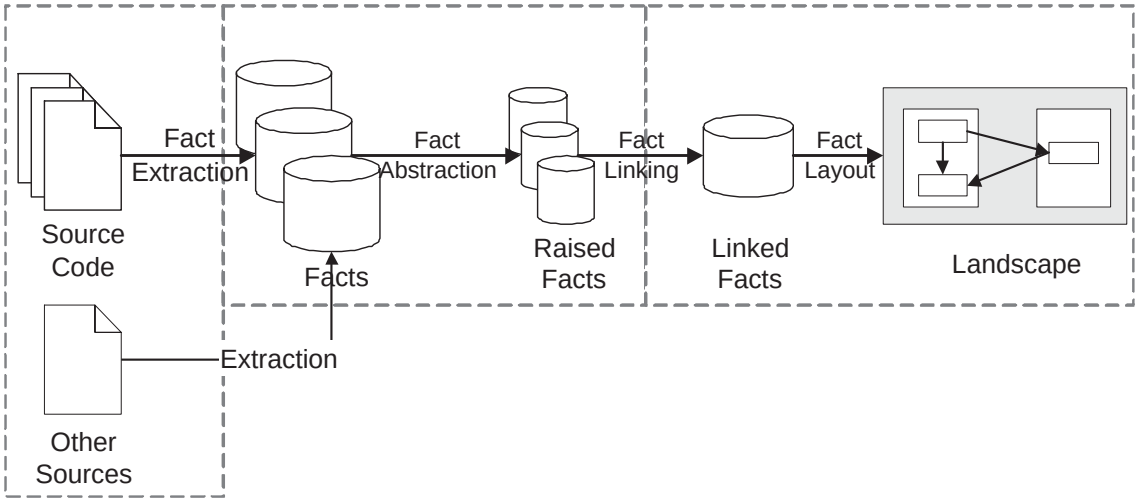

Figure 2.2: Swagkit's Reverse Engineering Process

2.2 The Reverse Engineering Process

We consider the reverse-engineering process to include the three stages mentioned in section 2.1 structured as a pipeline (Figure 2.1). When the process is implemented in tools such as the Swagkit, each of these stages are further refined. Figure 2.2 shows the actual plugins Swagkit uses during the reverse engineering of a concrete architecture from a source code base.

2.2.1 Fact Extractors

Fact extractors work by running the source code through a scanner and parser, generating an Abstract Syntax Tree (AST). This tree is then decorated with semantic edges, produced by a semantic analyzer, resulting in an Abstract Semantic Graph (ASG). The fact extractor can then manipulate this graph to produce facts at a particular level of detail. CPPX for example, uses the GCC [GCC] front end and then translates the facts to its own, slightly less detailed representation (Section 2.4.1).

The artifacts produced by a fact extractor should adhere to some syntactic format if they are going to be used for further processing. Currently there is a great deal of research in the Software Engineering community geared towards a standard exchange format for these artifacts [GXL, HSW00]. Some forms include TA [Hol97], GXL [GXL, HSW00], and RSF [MOTU93]. As well as a syntactic form, the artifacts also have semantics expressed by the constraints imposed by the schema. Schemas define the entity types and the relationship between them. For example, in a standard C++ schema, instances of the "Class" entity would be allowed to contain "Methods". Published schemas include Datrix [HHL+00], Columbus [FAB02] and Dagstuhl Middle Level Model (DMM) [LPT+01]. Unpublished schemas include Swag AST Schema (SAS) (Section 2.4.1), Swag Middle Level Schema (SMS) (Section 2.4.2) and the Swag High Level Schema (SHS) (Section 2.4.3).

Fact extractors exist for many languages including C [CNR90, MDH01], C++ [GC90, MDH01], Java [BG99] and web scripting languages such as VBScipt [HH02]. Run-time relationships can also be extracted using tools such as Third Eye [LRY00] and combined with the static relationships [RR02]. Files and modules can then be clustered into higher

level subsystems [TH00, MMCG99].

2.2.2 Fact Abstraction

Parsing is a hard problem and writing fact extractors for complicated languages such as C++ is a complicated engineering task [SHE02]. Sim *et al.* proposed that a common set of robust and correct parsers should be built and shared within the community [SSW02]. For a parser to be built once and shared among the community, source artifacts must be extracted with a fine granularity as not to limit their uses. However, if facts are extracted with such detail, the databases grow very large and the model to which these databases adhere becomes very complicated. Specific reverse engineering tasks, such as architectural recovery or class interaction, must filter out the artifacts from the database that are not required. In the case of PBS and Swagkit , the low level facts are "hidden" [Hol98] and only the architectural relations are shown. In the case of a class dependency tool such as jGRASP [BI02], only the relationships between classes would be displayed. Relationships between low level entities such as method calls and variable accesses, would be "abstracted" [Hol99] to higher levels.

2.2.3 Presentation

Facts can be presented in many ways. Graph drawing tools such as Dotty [NK94] and SVG [KWM] are commonly used to represent programs as box-and-arrow diagrams. Hierarchical editors such as the Landscape Editor [Pen93], `SHriMP` [SM95], and TANT [Lee00] are used to show the interaction between the entities and the containment structure simultaneously. However, visualization is not the only representation of program

artifacts. Facts could be stored as a database and queries could be used to retrieve the information requested. The facts could also contain links back to the original source code which could be resolved through tools such as `sgrep` [BTMG02].

2.3 The Representation of Extracted Facts

Data extracted from source code using reverse engineering techniques are commonly stored in various forms including: *1) marked up source code*; or *2) Entity-Relation models*. Marked-up source code has been used by Paukki *et al.* [PSK96], Cross and Hendrix [CH95], Cox and Clarke [CC01], Cordy *et al.* [CSDM01] and others, and consists of adding semantic information to the lexical source code through attributes and tags. Variables and other source entities can be marked with unique identifiers [CSDM01] so all references to the same entity can be tracked. To return to the original source, the added tags are removed.

Entity-Relation models [Che76], as means to represent software, have been used for years dating back to control flow and data flow graphs. More recently UML and Architectural ”Box and Arrow” diagrams have started to emerge. These models represent the source artifacts by a set of entities, connected to one another by a set of relationships. The ways in which entities can be connected include **entity A calls entity B**, **entity C inherits from entity D**, or any other relationship that may exist. The entities and relationships that are represented in the model are decided by the designer of the extractor. The models can be as sparse or as detailed as the designer wishes to specify it, although, if the models are to sparse then it may be useless and if the models are to detailed it may produce an incomprehensible amount of data. In order to recover the original source

from which the facts where derived, a large amount of data must be extracted including comments and formatting information such as spacing and indentations.

For the purpose of this thesis, we will consider the entity relationship model for representing software. We have chosen to do this because we feel this method is easier to present and we can make use of the current techniques for manipulating graphs such as relational calculators and SQL databases. Although we have chosen the entity relationship model, this choice doesn't limit us since it is possible to convert from entity relationship format to source code to marked-up source if the proper data has been extracted and a set of adequate parsers written.

2.3.1 Introduction to Graphs

Graph theory is the study of mathematical objects known as "graphs". For our work the Data Models are represented by directed typed graphs, graphs in which each edge belongs to a set specifying its "type" or "colour". Directed graphs can be used to represent such things as traffic flow, electrical networks and even models of computer programs. Before we begin explaining how this is done we need to provide some basic definitions used in graph theory. We follow standard terminology by Swamy and Thulasiraman [ST81]. The definitions A.1 to A.8 found in Appendix A "Basic Graph Theory Definitions" on page 131 will also be used repeatedly.

Definition 2.1: *A **Graph** $G = (V, E)$ where V and E are finite sets:*

- *V is a finite set of elements called **vertices***
- *E is a finite set of elements called **edges**, each of which is a pair (v_1, v_2) with $v_1, v_2 \in V$.*

Note that by assigning a "type" i to each edge E_j in E, a graph is a typed graph. This notion of typed graphs to aid in reverse engineering was introduced by Holt [Hol96].

Definition 2.2: *A **Typed Graph** $G = (V, R)$ where V and R are finite sets:*

- *V is a finite set of objects called vertices*
- *R is a finite set of relations. Given a set of vertices V and relations $R_1, R_2, ..., R_n$ (of colours) 1,2,...,n each relation R_i consists of a set of **ordered** pairs of vertices: $R_i \subseteq V \times V = V^2$. We define the notation xRy to mean that (x, y) is in relation R.*

The definitions to follow can either be applied to a single relation R_i or all the relations R. It is also important to notice from our definition of a typed graph (Definition 2.2) that edges have a direction and the identity relation (Definition 2.6) is permitted.

Definition 2.3: *The **Inverse Relation** $R^{-1} = \{(x, y)|yRx\}$.*

Definition 2.4: *The **Domain** of relation R with ordered pairs (x_0, y_0), (x_1, y_1), ..., (x_n, y_n) in graph G is the set $\{x_0, x_1, ..., x_n\}$*

Definition 2.5: *The **Range** of relation R with ordered pairs (x_0, y_0), (x_1, y_1), ..., (x_n, y_n) in graph G is the set $\{y_0, y_1, ..., y_n\}$*

Definition 2.6: *The **Identity Relation** $id(V_i)$ is $V_i \times V_i$ given $V_i \subseteq V$.*

Definition 2.7: *The **Composition** of relation $R1$ and $R2$ is $R1 \circ R2 = \{(x, y) \mid \exists z: xR1z \wedge zR2y\}$.*

Definition 2.8: *$R+$ **or Transitive Closure** is defined as all the pairs $\{(x_0, x_n)|x_0Rx_1 \wedge x_1Rx_2 \wedge ... \wedge x_{n-1}Rx_n$ where $n \geq 1\}$.*

Definition 2.9: *$R*$ **or Reflective Transitive Closure** is defined as all the pairs $\{(x_0, x_n)|x_0Rx_1 \wedge x_1Rx_2 \wedge ... \wedge x_{n-1}Rx_n$ where $n \geq 0\}$.*

Inherit	ClassA	ClassB
Contain	ClassB	Function_1
Contain	ClassC	Function_2
Call	Function_1	Function_2

Table 2.1: Representing a Typed Graph as a List of Tuples

Definition 2.10: *The **Forward projection** of set **s** through relation **R** is* $s.R = \{t | s \subseteq S \wedge (s,t) \in R\}$.

Definition 2.11: *The **Backwards projection** of set **s** through relation **R** is* $R.s = \{t | s \subseteq S \wedge (t,s) \in R\}$.

We consider the edge-relation as elements of a relational algebra, as advocated by Holt [Hol96], Feijs / Krikhaar / Ommering [FKO98] and others, for reverse-engineering.

2.3.2 ER Data Models for Reverse Engineering

Directed graphs are used to model computer programs by using the vertices to represent entities within a program such as *procedures*, *classes* or *subsystems* and the edges to represent relations between these entities. Examples of relations include: *calls*, *inheritance* or *containment*. The graph is often represented as a list of triples such as in Table 2.1 or pictorially as in Figure 2.3. *Rigi Standard Format* or RSF is a common exchange format for graphs of this type.

2.3.3 Schemas

Schemas can be defined to express constraints on a typed graph. For example, a schema for C++ will allow *Class* entities to *inherit* from one another, but will not allow a *Func-*

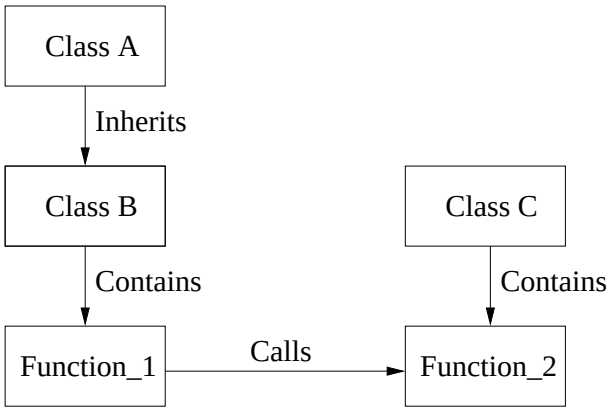

Figure 2.3: Sample Typed Graph

tion to *inherit* from a *Class*. Every node in a graph can be given a "type" by relating the node in the graph to a type defined by its schema. The schema defines which types of entities may exist in a graph and defines how they can relate. A graph which conforms to the constraints imposed by a schema is said to be an instance of that schema.

Schemas can also be defined as typed graphs G' and we use the Typed Graph definition above (Definition 2.2) to define schema graphs. We use V' to mean the set of schema vertices and R' to mean the set of schema relations. Constraints are specified by defining which vertices can be related. Graphs are then related to schemas through two relations I and Γ. I is the Instance relation, to mean an entity in the data is an instance of an entity in the schema. Γ is used to associate schema relations to data relations.

Figure 2.4 demonstrates this pictorially by showing the schema above the dashed line and the instance data below. The instance relation, I is represented by the dotted lines.

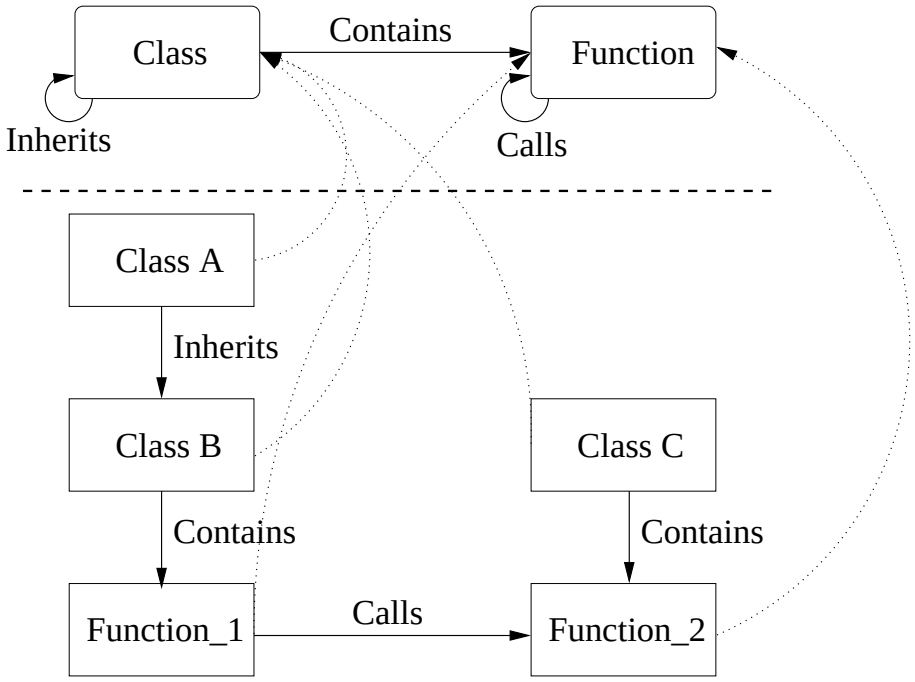

Figure 2.4: Typed Graph with a Schema

2.3.4 Schema Conformance

Schemas not only define what entities and relations can exist in the data, but they also define what entities can be related to one another. If, for example, the data had a reference from the function “main” to the variable “currentDate”, then the schema must allow functions to “reference” variables.

In order to determine if a graph conforms to a schema, the following terms need to be defined:

- G is a graph representing the data.
- G' is a graph representing the schema.
- I is the data-schema instance relation.

- If G is an instance of G' then Γ associates relations in the typed graph G to the relations in the typed graph G'. From now on r will always be a relation in the instance graph G and r' will always be a relation in the schema graph G' such that the following is true: $r' = \Gamma(r)$.

The instance graph G is said to conform to the schema graph G' if the following conditions are met: (For notation see 2.3.1)

Every entity type in the instance data must be present in the schema.

$$E.I \subseteq E'$$

Schema Conformance Condition 1

Every relation in the instance data must be present in the schema and for every relation, valid vertices are related.

$$r \subseteq I \circ r' \circ I^{-1}$$

Schema Conformance Condition 2

2.3.5 Hierarchical Constraints

As mentioned in the presentation section (Section 2.2.3) the notion of using hierarchical data to represent software structure has been realized by several different researchers [Pen93, SM95, Lee00]. Figure 2.5 (a) shows a hierarchical or tree-like breakdown of a hypothetical scanner / parser. In order to represent data in a form which can be abstracted we insist the typed graphs be structured as a collection of trees. This is a reasonable constraint since facts are often extracted as Abstract Syntax Trees, which are inherently hierarchical.

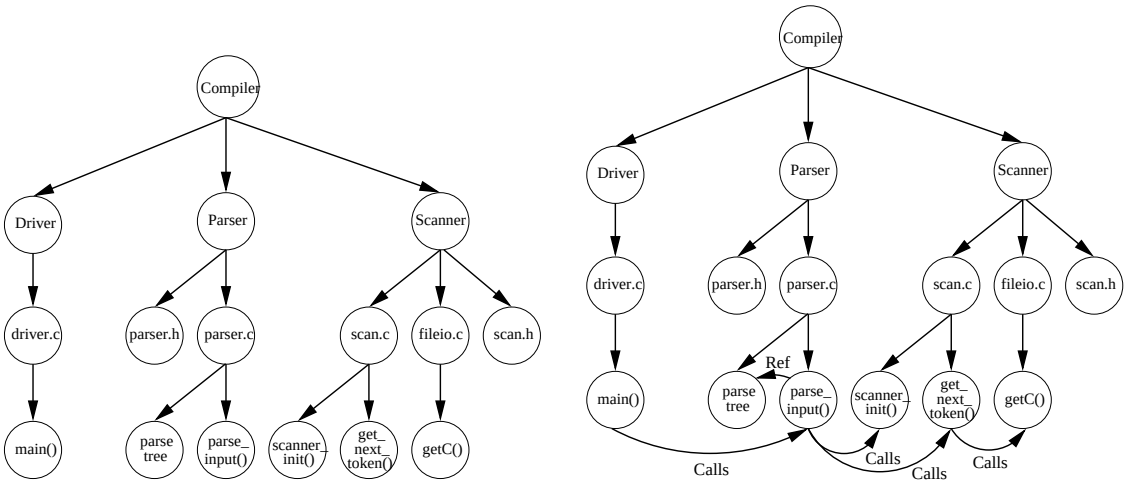

Figure 2.5: Hierarchical breakdown of a Parser / Scanner

We impose the hierarchical constraint by adding additional information to both the edges and the vertices in the graphs. Two types of vertices: **1) root vertex** and **2) non-root vertex**, and two types of edges: **1) structural** and **2) non-structural** are defined. Non-structural edges can extend in a non-tree like fashion in the instance graph G. Figure 2.5 (b) shows the scanner / parser example extended with non-structural edges. In order to describe our "forestness" we insist that each entity be reachable by exactly one root through a single path of structural edges. This means that if the non-structural edges were ignored, the instance data used to represent the software would be structured as a collection of trees.

2.3.6 Exchange Formats

In order to share data between reverse-engineering tools both the structure of the data and the schema to which the data conforms to must be agreed upon [HSW00]. Currently

SCHEMA	TUPLE	:
Inherit	Class	Class
Contain	Class	Function
Call	Function	Function
FACT	TUPLE	:
$INSTANCE	ClassA	Class
$INSTANCE	ClassB	Class
$INSTANCE	ClassC	Class
$INSTANCE	Function_1	Function
$INSTANCE	Function_1	Function
Inherit	ClassA	ClassB
Contain	ClassB	Function_1
Contain	ClassC	Function_2
Call	Function_1	Function_2

Table 2.2: Typed Graph as Triples with a Schema

formats such as TA, GXL and RSF are commonly used for the representation of the source code as typed graphs. Ferenc *et al.* [FSH+01] proposed that a standard schema be created for such languages a C++ so as to ensure a common description of the form of the instance data. TA [Hol97] and GXL [HSW00] are flexible exchange formats which do not prescribe a schema but rather provide features to allow a software engineer to specify his own schema. TA and GXL combine both the schema graph G' and the instance graph G' into a single graph. TA would represent the model shown in Figure 2.3 and Table 2.1 as the factbase in Table 2.2.

In the case of TA the instance relation I is represented in the data as the "$INSTANCE" relation and Γ is an implicit relation based on the spelling of the relation names. If an relation R_i in the instance graph G has the same spelling as a relation R'_i in the schema graph G' then $R'_i = \Gamma(R_i)$.

Our models can also be extended to allow for the assignment of attributes to the

entities. This allows information such as a *line number* or a *file name* to be associated with source code entities. Each attribute that can be attached to an entity should be defined in the schema.

2.4 Exchange Schemas

In order to allow for the sharing of data between tools, standard schemas must be adopted [FSH+01]. At Dagstuhl in 2000 [DAG], a workshop was dedicated to the creation of three standard schemas: *AST Level, Middle Level* and *Architectural Level*. Schema standardization and promulgation is difficult. At the lowest level (AST Level), schemas are complicated and it's difficult to adapt existing tools to new complex interfaces. It was equally as challenging to standardize on a single architectural model since different researchers have different views on what an architectural model should represent [GS93, PW92, Kru95, Hol02, KBWA94]. The only schema which emerged as a direct result of this workshop was the Dagstuhl Middle Model. This section will explore three common schemas used in the Software Architecture Group at the University of Waterloo.

2.4.1 Swag AST Schema (SAS)

The *Swag AST Schema* is based on that of the Datrix project [HHL+00] at Bell Canada [Bel]. The output of CPPX conforms to the Swag AST Schema and this schema can be used to model C or C++ programs. There are 66 entities and 23 relationships defined in the schema. Data which conforms to this schema is to detailed to be easily used and the data must be abstracted first. The `hello world` program, made famous by Kernighan

	C Version	C++ Version
Lines of Pre-processed Source Code	282	16,551
Size of Factbase	82 Kb	4.6 Mb
# Of Relations	10	26
# Of Entity Types	19	54
# Of Entities	751	49,380
# Of Edges	1,532	77,808
# Of Entities Per Line of Code	2.7	3.0
# Of Edges Per Line of Code	5.4	4.7

Table 2.3: Summary of CPPX Output for `Hello, World!`

and Ritchie [KR88], represented in the SAS notation, results 82 Kilobytes of TA (383K of GXL) and the following one written in C++ produces 4.6 Megabytes of TA (18M of GXL). This is partially because after preprocessing the simple program expands to 16,551 lines of code. It is this preprocessed source code that is passed through the fact extractor to produce such a large model. Furthermore the schema requires on average about three entities and five relations for every line of post-processed source code. Table 2.3 summarizes these results.

```
#include <iostream.h>

int main() {
    cout << "hello, world!" << endl;
    return 0;
}
```

Entities	Relations
cSubSystem	cRefersTo
cCompliationUnit	cLinks
cModule	cInstance
cFile	contain
cNameSpace	
cFunction	
cObject	
cLiteral	
cType	
cAggrType	
cEnumType	
cAliasType	
cEnumeration	
cScopeGlb	
cScopeCompil	

Table 2.4: SWAG Middle Level Model

2.4.2 Swag Middle Level Schema (SMS)

The *Swag Middle Schema* was defined to allow CPPX users to browse the CPPX artifacts in a more abstract form. The Swag Middle Level Model has 4 relations and 15 entities. The entities and relations are listed in Table 2.4.

2.4.3 Swag High Level Schema (SHS)

The *Swag High Level Schema* was defined to show only high level entities geared towards understanding the architecture of a software system. It includes “Subsystem” and “Module” entities and contains the relations “Calls”, “Uses”, “ImplementsBy” and “Contains”. Figure 2.6 shows the Swag High Level Model as a diagram. The solid “Contain” edge is considered the structural edge and the double octagon “Subsystem” entity is considered

the root vertex. From this graph it can be seen that “Modules” are permitted to “Call” one another, “Use” one another, or be “ImplementedBy” one another.

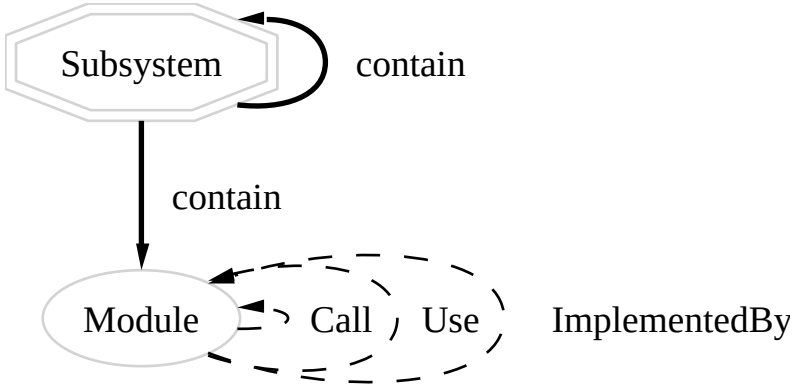

Figure 2.6: Swag Architectural Level Model

2.5 Summary of Background

The field of software reverse engineering is growing as software becomes more valuable and maintenance activities become harder to perform. The standard reverse-engineering pipeline consists of artifact extraction, artifact abstraction and artifact presentation. The data models used for the pipeline are generally of the Entity-Relation types with entities being software artifacts and relations being relationships between these artifacts as expressed in the source code. By formalizing the schema, the data can be published and transfered from one reverse-engineering tool to another. These fact bases can consist of tens of thousands of relational pairs for trivial programs because so much detail is extracted at the AST level. To make these fact bases usable the data must be abstracted. The next chapter presents actual abstractions that are performed by current reverse-engineering tools.

CHAPTER 3

Abstractions for Reverse Engineering

A little inaccuracy sometimes saves a ton of explanation.

– Hector Hugh (H.H.) Munro (Saki)

As systems increase in size, and source code extractors extract more data for every line of source code, the artifact database grows too large to be understood. It becomes harder to navigate the extracted facts than it does to navigate the source code. For this reason, reverse engineering must both: *1) extract system artifacts* and *2) abstract the artifacts.* This chapter presents abstractions performed by reverse engineering tools to help filter out extraneous information.

This chapter starts by looking at what an abstraction is (Section 3.1) and how it can assist a software engineer to recover the design of a large system (Section 3.1.1).

Section 3.2 presents a review of current abstraction techniques performed by reverse engineering tools. The tools studied include Rigi, PBS, CIA and Dali. Section 3.3 creates a classification based on the affect the abstraction has on the underlying schema to which the data it's abstraction conforms. The chapter then proceeds to show a set of schemas which can be used to answer specific questions about a software system (Section 3.4). A formal definition of abstraction is given (Section 3.5) along with a few theories which will help when applying the abstractions in practice. Finally, the chapter concludes with a summary of abstractions for reverse engineering (Section 3.6).

3.1 What is an Abstraction?

There are over 2.5 million relations in the factbase for the Pine Mail and News Reader presented in section 1.1. This factbase, presented as is, cannot be processed very easily by a software engineer, or other reverse engineering tools. Figure 3.1 however, shows an architectural diagram from Pine[1]. This diagram was produced from the factbase directly and can be used to help organize a team of developers, bring a new developer up to speed on the project [SCHC99] or assist with architectural repair [TH99].

Webster's Revised Unabridged Dictionary defines abstraction as "The process of leaving out of consideration one or more properties of a complex object so as to attend to others [DIC]." This definition is similar to a projection or view in a database system. Abstractions also deduce new information when leaving out of consideration other information. For example, in the Pine factbase there is no direct relation between

[1]Pine was extracted using CPPX [MDH01], abstracted using grok [Hol] and clustered using Bunch [MMCG99]

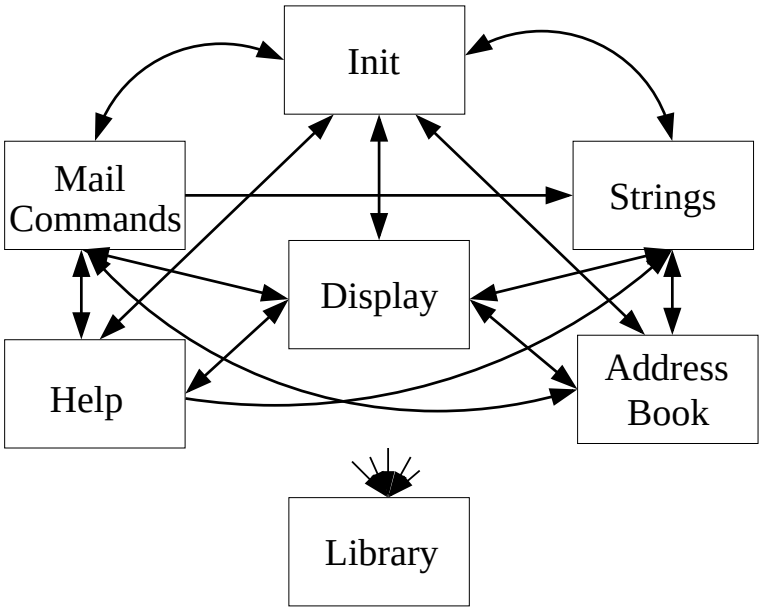

Figure 3.1: Pine Architectural Diagram

subsystems, but rather these are deduced by looking at how the functions and objects within each subsystem interact. We consider abstraction to be both 1) Projection, and 2) Deduction.

When large amounts of data have been extracted it is difficult to present all the information to a user in a way that is meaningful. Abstractions are performed for three reasons:

- *Space considerations:* A factbase for a large program may not fit into the computers memory. For example, if a 12 million line program (after preprocessing) was extracted resulting on average in, three entities (each requiring 128 bytes for the name and attributes) and five relations (each requiring 4 bytes) for every line of code, $1.2 \times 10^7 \times ((128 \times 3) + (4 \times 5)) = 4.8 \times 10^9$ bytes would be required, more than can be addressed in a 32-bit address space.

- *Time considerations:* Algorithms to query the data take a long time on data that has not been abstracted. For example, during the process of program understanding, it is common for software engineers to address the problem in a a top down fashion, by first looking at how Subsystems, Modules and Classes interact. Programs on the other hand, represent this data near the leaves of the trees, connecting literals, functions, and statements. While browsing the high level structure, the vertices at this level should contain the relationships of their descendants. Otherwise, the query engine will have to compute these relationships for every query performed.

- *Cognitive considerations:* Data that has not been abstracted does not match a software engineers mental model of the system. For example, software engineers will often talk about how *classes interact*. In most programming languages however, there is no direct interaction between classes other than inheritance. Software engineers often consider the interaction of methods and variables contained within a class to mean the interaction of the classes.

By filtering data by means of an abstraction, future queries can be simplified because of the three reasons listed above. It is important to take careful consideration when abstracting as to what data is kept and what data is "abstracted" away.

It may be argued, using Moore's Law [Moo65], that spatial and temporal reasons become less important as the speed and memory increase, however, if the number of entities can be reduced by a factor of ten to a hundred, spatial and temporal reasons will remain important. Also, as the hardware we can use to analyze programs on increases, the size of the programs we are analyzing also grows. The complete fact base for a huge system (10^7 Lines of Code) cannot be properly examined due to memory and time considera-

tions. These problems can only be solved if the system can be abstracted in parts and the parts assembled to form the entire system. In programming environments, linking is the process that turns a collection of program fragments into a runnable program [Car97]. In a reverse-engineering framework, linking is the process of assembling program fragments that have been reverse engineered into a completely reverse engineered system. In order to allow abstraction to take place before or after the linking step, abstraction must commute with linking. This will be addressed in Section 3.5.4.

Abstraction as a technique for cognitive assistance is the most important reason for abstracting. If a software engineer is studying the software architecture, it makes sense to remove entities that do not contribute to the architecture. Entities such as expressions and local variables can be hidden to help remove clutter. However, if the software engineer is studying data access, local variable and expressions become the core information and subsystem hierarchy can be ignored.

Also, for cognitive reasons, abstractions must be easy to specify. The user should be able to specify abstractions without prior knowledge of the system and refine the abstractions as more is learnt. This is because abstraction is the first interactive step in the reverse engineering framework and often a user will reverse engineer a system to gain greater insight into how it is built. If a user is required to have extensive knowledge about the system, the need for the reverse engineering activities will be negated.

3.1.1 How can Abstractions Assist Program Understanding?

Reverse engineering is the process of extracting and abstracting artifacts from a software system. Program understanding means building a knowledge base from these artifacts

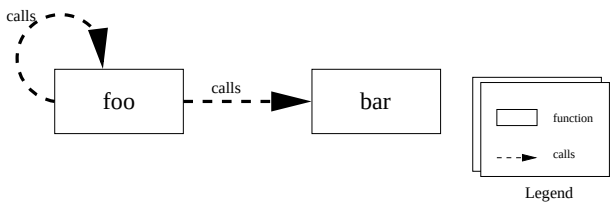

Figure 3.2: Call Graph

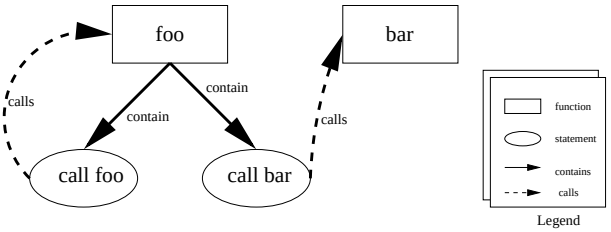

Figure 3.3: Actual Representation of a Call Sequence

that represents the program, and recognizing program fragments as components in the knowledge base [MJN+92]. What program fragments are represented in the knowledge base and how these program fragments are related has never been formally defined and furthermore cannot be formally defined because software engineers have different requirements for a knowledge base depending upon the activity they are performing.

For example, it is common for software engineer to inquire about the call graph for a program and "hide" the actual statements found within each function. This results in a diagram as in Figure 3.2 where the boxes represent functions and the arrows represent calls. However in a program, functions don't actually have references to other functions rather their statements may have references to functions (Figure 3.3). This view is equally valid and may be required if a software engineer wishes to located the line of code on which the call occurs. In order to generate Figure 3.2 from Figure 3.3, the call edges from `foo` to `bar` and from `foo` to itself must be deduced from the lower level call edges.

It is tempting to consider abstractions to target particular levels in a program. This has even been addressed by creating schemas which focus on such levels as: *1) AST Level* (includes all information in the AST), *2) Middle Level* (includes information down to functions / classes), *3) Architectural Level* (only includes subsystem and module interaction). However, abstractions that span levels are also very important. One example of this is the removal of all program entities except “Subsystems”, “Types” and “Classes”, and relationships between these elements. This abstraction would result in a factbase which represents the type system for a piece of software clustered into a set of subsystems, describing how the types are related. Each subsystem could be given to a different developer to analyze further. This spans levels since the final factbase contains some high level entities such as subsystems, and some low level entities such as types and variables.

The different ways that information can be combined, removed and deduced from other information is the motivation for the study of current abstraction techniques in reverse engineering tools.

3.2 The Use of Abstraction in Current Reverse Engineering Tools

The use of abstraction to make a system easier to understand is a phenomenon many researchers have used without noticing and therefore without standardizing. This section attempts to review the approaches taken by different reverse engineering tools in order to abstract the information and produce different views. For this section, abstraction will be considered any filtering operation performed on the data. A formal definition of

abstraction will be presented in Section 3.5. The tools studied include PBS [FHK+97] and Swagkit [Sof] , Rigi [MK88], Dali [Dal] and CIA / CIA++ / Cia [CNR90, GC90, CFKW95].

3.2.1 PBS and Swagkit

Overview

The Portable Bookshelf (PBS) is a web-based paradigm for the presentation and navigation of information representing large software systems [PBS]. PBS allows the user to navigate software systems at the architectural level by visualizing subsystem and file interaction. It has been used to visualize the architecture of Linux [BHB99], Mozilla (Netscape) [GL00], XFig, VIM, GCC [TG02] and many other unpublished student projects. Swagkit [Sof] has replaced PBS at the University of Waterloo but uses much of the same technology as its predecessor, including the relational calculator `grok` [Hol].

Structure

Swagkit and PBS have four major steps:

1. *Extract:* Produces a factbase for every compilation unit in the system using CPPX or CFX.

2. *Prepare:* Lift each of the individual factbase to create a higher level view.

3. *Link:* Link the individual factbase to produce on factbase for the entire system.

4. *Layout and View:* Layout the entities and visualize them using a Landscape Editor.

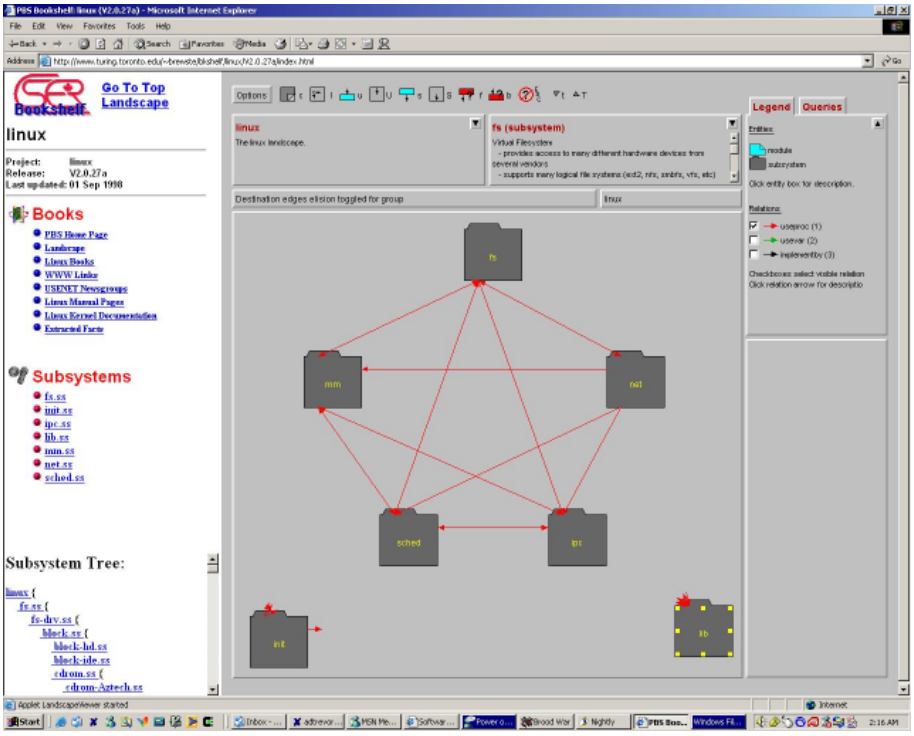

Figure 3.4: PBS Visualization Using Lsedit

The architectural diagram for Swagkit was presented in Chapter 2, Figure 2.2 on Page 13.

Display Methods

Swagkit and PBS both use the Landscape Editor [Pen93] `lsedit` to present their results. The landscape editor allows a software engineer to navigate architectural models across different levels of hierarchy. Figure 3.4 shows a visualization of the Linux kernel using `lsedit` [BHB99]. `Sgrep` [BTMG02] can also be used to query the factbase for low level relationships by specifying high level entities.

PBS and Swagkit's Uses of Abstractions

PBS and Swagkit extract low-level facts by means of a C extractor called `cfx` and a C++ extractor called CPPX respectively. High-level facts are then deduced from the low-level facts. Deduction is implemented using a relational calculator called `grok`. `Grok` works by creating a representation of the factbase in memory as an Entity-Relation model. `Grok` then provides operations which affect the entire graph, not just a single entity [Hol]. An example of a deduction PBS and Swagkit perform is lifting the low-level facts to the file level. After the file-level facts are deduced, the low level facts are removed.

The use of a relational calculator to deduce high level facts from low level facts is similar to the work presented by Feijs, Krikhaar and Van Ommering [FKO98]. A relational calculator determines interactions between subsystem descendants and then elides the interactions to the subsystem level. Elision is done by looking at interactions between the descendants of a set of nodes. If two descendants of two different entities have a relation then it is said that the two entities have this relation. The low level information can then be discarded without losing the subsystem interaction. The level at which the abstraction takes place can be varied by specifying a different schema vertex. Some common levels include "File Level", "Function Level", "Class Level" or "Package Level". Of course some of these only make sense in the context of a particular programming language.

3.2.2 Rigi

Overview

Rigi is an interactive, visual tool designed to help you better understand and re-document your software [RIG]. Rigi was built at the University of Victoria, under the direction of Hausi Müller, his colleagues and students starting in the early 1990's.

Structure

Rigi consists of two parts, Rigi-parse and Rigi-edit. Rigi-edit provides editing features for the data which reside in the underlying database. Rigi-parse parses the source code and fills the data repositories. To help manage the complexity of the graph, Rigi-edit allows users to collapse related artifacts into subsystems (both manually and automatically). The collapsed system can then be navigated, analyzed and presented using different layout algorithms.

Display Methods

Rigi originally used Rigi-edit and currently uses Simple Vector Graphics (SVG) to present the data. Both of these visualization methods provide a set of basic editing features, some of which will be discussed in the next section. Figure 3.5 shows how Rigi uses SVG to represent an entity relationship diagram [KWM].

Rigi's Uses of Abstractions

Rigi's abstractions allow a collection of related modules to be grouped together focusing on attention on the groups instead of the individual objects.

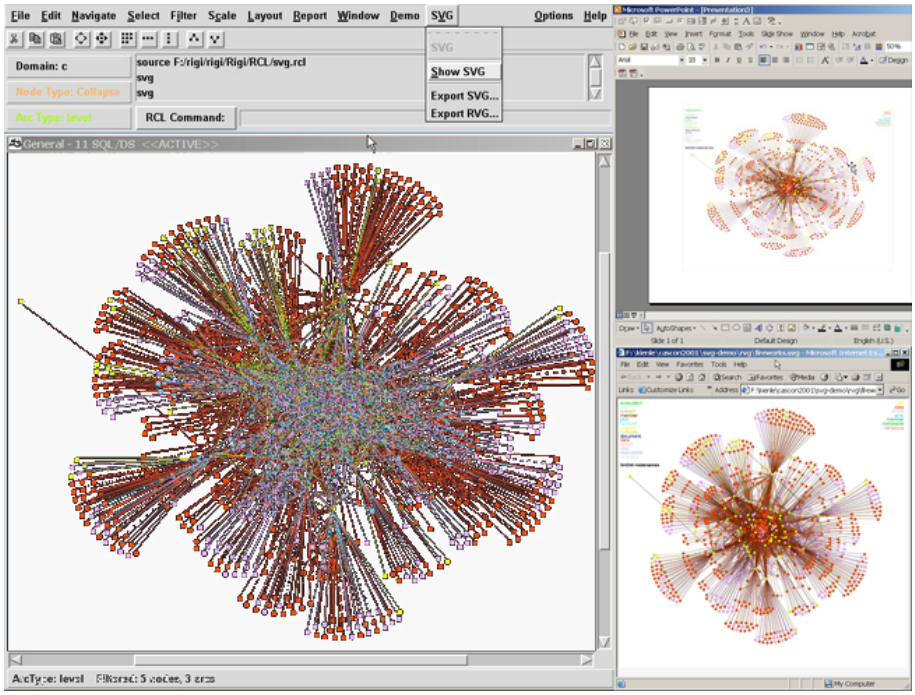

Figure 3.5: Rigi Visualization Using SVG

Command	Purpose
Cut	Remove an entire subgraph from the model and place it in a buffer.
Copy	Copy an entire subgraph into a buffer.
Paste	To copy the contents of the buffer into the model.
Zooming	Impose a more detailed view of part the system.
Filtering	Filter entities based on a selection criteria to impose a different view on the entities of the repository.
Grepping	String search using regular expressions
Collapse	Replaces a subgraph by a single node.
Remove Omnipresent Nodes	Remove nodes which have a high degree of interconnectivity with other nodes in the system.

Table 3.1: Standard Rigi Commands

> *To manage the complexity of the graph, an editor allows you to automatically or manually collapse related artifacts into subsystems. These subsystems typically represent concepts such as abstract data types or personnel assignments. The created hierarchy can be navigated, analyzed, and presented using various automatic or user-guided graphical layouts.* [RIG]

Rigi provides a set of basic operations and some more complex operations built on top of these. Pre-defined scripts are also provided that make use of this set of commands to transform the graph in common ways. An example of a common transformation Rigi supports is creation of a call graph from a fact repository. The user is free to use the pre-defined scripts or define their own transformations using the commands provided. Commands include both data filtering and data-producing operations. A list of the operations has been presented in Table 3.1.

Rigi provides an assortment of ways in which nodes can be grouped into subsystems. These groupings are not the focus of this thesis and are not listed. There is one notable exception and this is Rigi's notion of *Removing Omnipresent Nodes*. This is a very interesting filtering operation since it removes the nodes that appear everywhere. The primary reason for this operation is to remove nodes which are highly used such as drivers or libraries. The use of these nodes is often better represented as a piece of prose instead of cluttering the graph with additional edges.

3.2.3 Dali

Overview

Dali is a workbench used to reconstruct the architecture of a software system [Dal]. The Dali Architecture Reconstruction Workbench was developed at the Carnegie Mellon Software Engineering Institute under the guidance of Rick Kazman, Liam O'Brien and Chris Verhoef [KOV01]. Dali is able to extract source code artifacts, build abstractions from the extracted artifacts and generate architectural views. Dali has been used on a wide range of software system ranging in size from tens of thousands of lines of code to many millions lines of code.

Structure

Dali consists of 4 major steps, similar to the pipeline described in Figure 2.1

1. *View Extraction:* Extracts program views using various technologies including: parsers, abstract syntax tree-based analyzers, lexical analyzers, profilers, and code instrumentation.

2. *Database Construction:* The different extracted views are converted to the Dali format and stored in a relational database.

3. *View Fusion:* Queries are performed to manipulate the different extracted views to create *fused* views.

4. *Architecture Reconstruction:* The user is provided with tools to visualize, explore and manipulate the views. Architectural reconstruction also provides pattern definition and recognition facilities to help the software engineer recognize architectural patterns.

These steps are documented by Kazman *et al.* in the *Architecture Reconstruction Guidelines* [KOV01].

Display Methods

Dali uses Rigi edit to display entity relationship diagrams. Dali also supports direct manipulation of the underlying database for analysis purposes. This allows a software engineer to produce his own customizable view from the data for further processing.

Dali's Uses of Abstractions

View extraction and database construction (steps 1 and 2) are considered fact extraction in the reverse engineering reference architecture (Section 2.2). View fusion and architecture reconstruction (steps 3 and 4) are the fact manipulation or abstraction and presentation phase. However, Dali does not perform the architecture reconstruction (or abstraction phase) automatically. It assumes a tight loop between the architecture recon-

struction and the analysis phase. Dali provides mechanisms for the software engineer to perform custom abstractions.

Dali does not perform any abstraction before presenting the user with the artifacts. The system allows users to abstract the data themselves in order to highlight particular information. Users will produce custom views based on relational queries. The views can be interpreted, refined, or rejected based on the needs of the software engineer. Dali assumes "there are no universal completion criteria for this process; it is complete when the architectural representation is sufficient to support the analysis needs of the Dali users" [KOV01].

Dali abstracts data by creating new nodes known as *aggregates* from combinations of one or more existing nodes. The old nodes can then be removed from the view. For example, if a user wished to combine a function (F) and its three statements (S1, S2 and S3) together, he would create a relational query to create four new entities (F+F, F+S1, F+S2 and F+S3). The original F, S1, S2 and S3 could then be discarded. These new entities could now be referred to as F+.

Dali supports the following user-driven abstractions:

- *Type identification.* This highlights all the types in a system for further processing.
- *Aggregate local variables with functions.* This combines local variables with their function entity.
- *Aggregate members with classes.* This combines member variables and methods with their class.
- *Compose architecture-level elements.* This combines modules such as classes with

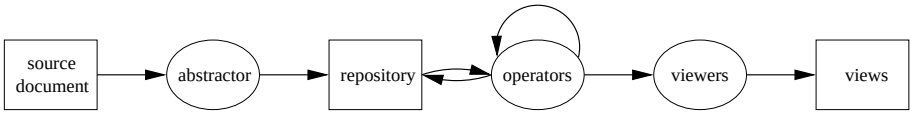

Figure 3.6: Ciao's Repository-Based Reverse Engineering Process

the subsystem they are contained within.

3.2.4 CIA / CIA++ / Ciao

Overview

The C Information Abstraction System (CIA) [CNR90], the C++ Information Abstractor (CIA++) [GC90], and the graphical navigation software Ciao [CFKW95] is a systems designed for analyzing C and C++. The system was developed at AT&T Laboratories.

Structure

AT&T's suite of tools are developed and packages as separate projects. The extraction phase is performed for C and C++ by CIA or CIA++ respectively. Like Dali, the abstraction and visualization phase is combined into a single tool. This tool, Ciao, allows the software engineer to perform abstractions, view the results, and then refine their abstractions. Figure 3.6 shows the high-level view of Ciao's "repository-based" reverse engineering process. Chen *et al.* claims the repository-based process allows various abstractions to be generated directly without repeated processing of the original software documents [CFKW95].

Display Methods

Ciao provides three classes of basic views for the software engineer to use: *1) relational view*, *2) source view* and *3) graph view*. The relational view shows the relations that exist in the system, the graph view represents these relations as a graph, and the source view allows the user to associate the database and graph view back to the source code. When using this environment, the software engineer is first formulates a query by specifying constraints on attribute values. The query will produce a set of entities and relations which can be presented to the software engineer as either a source view, relational view or graph view [CFKW95]. These queries can then be refined or further constrained until the software engineer finds what he is looking for.

CIA / CIA++ / Ciao's Uses of Abstractions

Unlike the other tools studies, CIA includes a step before the artifact extraction. In this step a conceptual model for the target programming language must be defined. This conceptual model acts as a requirements specification for the extractor or the schema to which the extracted data must adhere to. This allows the abstraction to be done in the extractor itself. This method is less flexible since a new extractor has to be written in order to extract facts at a different abstraction level, however, the tool designers hope that a subtable level of abstraction is chosen such that interesting queries can be performed without a large overhead of extraneous information.

The current version of the C++ extractor (CIA++) tracks only global entities and the interaction between them [GC90]. The following entities are recognized:

- Files

- Macros
- Types
- Variables
- Functions

Relationships include: *include, inheritance, friendship* and *reference*.

As mentioned above, further abstractions can then be performed through Ciao, the visualization application for this suite of reverse engineering tools. The filtering operations done at this stage are similar to Dali's. In addition to most of the operations computable with Dali, Ciao provides two interesting operations clouser and focus.

- *Closure* computes all the entities and relationships that are reachable starting from a specified set of nodes. This can be useful when determining code reachability [CGK97].

- *Focus* shows the direct relationships between a single entity.

3.2.5 Summary Of Techniques

All reverse engineering tools recognize the need to abstract information in some form or another. Different tools provide different mechanisms in order to perform these abstractions.

The reverse engineering process has be architected so that abstractions occur during the second stage. Although in practice it can be seen that this is not always the case. CIA for example performs abstractions during the extraction phase and Ciao performs additional abstractions during the presentation phase.

Phase	**Tools**
Extraction	CIA / Ciao
Automated Abstraction	PBS
Manual Abstraction	Dali, Rigi, CIA / Ciao

Table 3.2: User Interaction with Abstraction

Abstractions are continuously being applied to filter the data in different ways throughout the reverse engineering-process. Tools such as CIA and Swagkit perform some abstractions before the entire database has been assembled. For this reason it is crucial that any operation which is considered an "abstraction" be commutable with linking, ensuring that the same result will the achieved regardless of when the abstraction occurs.

It has been noticed that abstractions which happen earlier on in the process usually remove artifacts from the database, whereas abstractions which happen near the end of the process, simply hide the information and allow the software engineer to revert to the original view. For the purpose of abstraction, it doesn't matter how the information is represented and how the filtered data is hidden.

We have also found that abstractions performed near the beginning of the reverse engineering process require less user input. This may most likely be attributed to the fact that the user cannot perform complicated abstracts early on, because he has limited knowledge of the system. Table 3.2 shows where each tool performs its abstractions.

The types of abstractions performed by each tool varies as well. Up to now we have considered abstractions to be any data filter operation, although, as we will explain in Section 3.5, an arbitrary filtering operation will not necessarily be an abstraction. Table 3.3 shows the types of filtering operations performed by each tool. These operations will be revisited in Chapter 4 and Chapter 5 during the abstraction pattern sections.

Tool	Abstraction Done
PBS	Lift Artifact Removal
Dali	Type Identification Aggregation Compose elements
CIA	Closure Focus
Rigi	Node Removal Node & Edge Type Filtering Collapsing Remove Omnipresent Nodes

Table 3.3: Filtering Operations Performed

These abstraction types can generally be categorized into *Data Abstraction* or *Schema Abstractions*. Schema abstractions are abstractions which change the data model by changing fundamental parts of the schema. Data models have no deterministic abstracting effect on the schema.

3.3 Categorization of Techniques

It is possible that when a factbase is filtered, the underlying schema will change. For example, if a software engineer were to remove all "variables" from a factbase there would be no need to represent a "variable" vertex in the schema. However, if a software engineer was to filter all entities whose name matched the pattern `UserInterface*`, this may result in some "variables", "types" and "functions" being removed, but other entities of these types remaining. We consider the first type of abstraction, an abstraction specified by naming schema vertices, a **schema abstraction**. We consider the second

type, an abstraction specified directly on the data, a **data abstraction**.

3.3.1 Data Abstraction Techniques

Data abstractions are abstractions that cannot be specified solely on the schema. The schema may change to include new edges after a data abstraction, but the number of entities are not uniformly reduced. Of course, the number of schema entities may still be reduced, if for example, all entities named `UserInterface*` were removed, and in the instance data the name of every variable matched this pattern. In this case there would be no need for a "variable" vertex in the schema anymore. We say the schema is not uniformly reducible because the affect of reducing the schema depends on the data.

After any data abstraction occurs, a refocusing operation is performed which ensures that the resulting data is still a forest with respect to its structural edges (Section 2.3.5). Since no new structural edges can be introduced during abstraction, the refocusing just has to ensure that all entities are still reachable from the root and remove any entities which are not.

3.3.2 Schema Abstraction Techniques

Schema abstractions are operations specified by using schema vertices and edges in the specification of the abstraction. Some examples of schema abstraction operations include: *remove all "Classes" in the system* or *elide all "Function" bodies to the "Function" node* After a schema abstraction occurs a refocusing of the schema occurs. This simply ensures that all entities in the schema are reachable through a path of structural edges from a root node (Section 2.3.5). If an entity is not reachable from a root node in

the schema then any instances of that entity in the data will not be contained within the tree. After this refocusing occurs, the same abstractions are applied to the data. Finally the data refocusing occurs, insuring that all data entities are still reachable from a root.

3.4 Abstracted Schemas

Through the use of schema abstractions, higher level schemas are derived from lower level schemas. For example, a middle level model (showing call relations, type information, class hierarchy and subsystem interaction) can be derived from an AST model (such as the Swag AST Schema). More specific schemas which focus on a single view such as the 1) call graph, 2) inheritance hierarchy, 3) type information or 4) concrete architecture, can be created using a combination of schema abstraction techniques. The mechanism for creating new schemas from more complicated schemas is done through the application of schema abstraction patterns (Section 5.1).

It is not the indent of this thesis to determine the usefulness of different schemas, although by looking at the current data models used in the reverse engineering community it seems that some schemas are more common than others. This section will briefly outline some common schemas that can be used to answer specific reverse engineering questions.

Class Level

Class interaction schemas show information about class interaction in an object-oriented system. This is very similar to the UML Class interaction diagram. The schema in figure 3.7 shows inheritance between classes as well as member variable and method

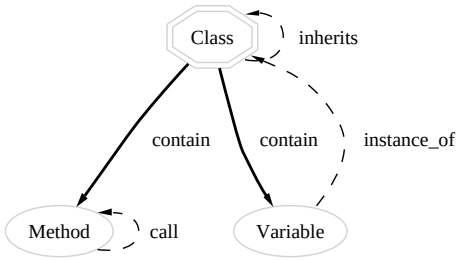

Figure 3.7: Class Interaction Schema

Figure 3.8: Function Interaction Schema

interaction. This schema can be used to answer questions such as, “How do the classes interact?” In each schema, the double octagon represents a root vertex and the solid arrows represent structural edges.

Function Interaction

Function interaction schemas model how functions in the system interact with one another. This schema can be used to determine function reachability and dead code analysis. Figure 3.8 shows a graphical representation for this schema.

Type Information

Type information schemas show where all the variables are used in a program and what type’s the variables have. This schema includes the entities: *1) Statement, 2) Variable, 3) Integer, 4) Float* and *5) Class*, and the relations *InstanceOf* and the structural relation

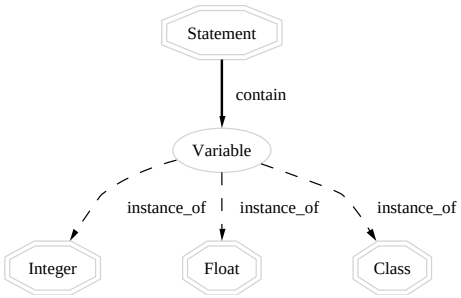

Figure 3.9: Type Interaction Schema

contain. Of course this could be extended to include other types. Figure 3.9 outlines this schema. This schema can be used to answer questions about how types are being used.

3.5 Abstraction Formalization

Before abstractions can be properly categorized and properties of abstractions listed, a formal definition of abstraction must be presented. Also it needs to be shown how two graphs representing different parts of a software system can be linked together. We do this because many of the reverse-engineering tools studied in Section 3.2 apply abstractions both before and after the linking stage and it must be ensured that the same result is obtained regardless. Therefore, our definition of abstraction must commute with the union operator “link”.

3.5.1 Definition of Abstraction

From our definition of abstraction in section 3.1, abstractions are either the removal of entities, the removal of edges or deduction of new edges. New edges will always be deduced from old edges at a lower level in the tree. Throughout this section we will use

the following notation:

- $T(v_1, v_2)$ will represent the typed directed edge relating the vertices (v_1, v_2), directed from (v_1) to (v_2).
- $(\overrightarrow{v_1, v_2})$ will represent a directed path from (v_1) to (v_2).
- $(\overleftarrow{v_1, v_2})$ will represent the inverse of a directed path from (v_2) to (v_1).

We will formally define an abstraction $\mathcal{A}$ as a graph morphing composed of zero or more operations of the following kinds:

- *Removal of a typed edge*: $e\mathcal{R}_{T(v_1,v_2)}$
- *Removal of a vertex*: $v\mathcal{R}_v$
- *Deduction of a new typed edge*: $e\mathcal{D}_{T(v_1,v_2)}$ such that a path exists from v_1 to v_2 of the following form: $\mathrm{C}^* \circ u \circ \mathrm{P}^*$, where C^* is the composition of 0 or more structural edges (descendants), u is a single non-structural edge and P^* is the inverse of 0 or more structural edges (ancestors). For the remainder of the thesis, the path $\mathrm{C}^* \circ u \circ \mathrm{P}^*$, will be considered a "lift".

An abstraction must be specified independent of any particular instance graph.

Because of this definition, not all filtering operations can be considered an abstraction. Namely the closure operation performed by CIA on page 47 is not an abstraction because the deduced "reachability" edge is not a considered a lift.

3.5.2 Definition of Linking

The linking $\mathcal{L}$ of two linkable graphs G_1 and G_2 is defined as:

$$\mathcal{L}\,(G_1, G_2) = (G_1 \cup G_2).$$

Two graphs G_1 and G_2 are said to be mutually linkable if:

- Both G_1 and G_2 are forests and
- $\forall v \in G_1 \cap G_2$ the unique path $(\overrightarrow{Root}, v) \in G_1$ is the same as the unique path $(\overrightarrow{Root}, v) \in G_2$.

3.5.3 Linkable Graph Property

If G_1 and G_2 are mutually linkable, then $\forall v \in G_1, \notin G_2$, any vertex x on the path from v to a leaf of G_1 is not contained in G_2.

Proof:

1. Assume $x \in G_1 \cap G_2$.
2. Since both G_1 and G_2 are trees, and x is on the path $(v, \overrightarrow{leaf})$, v must be on the path $(\overrightarrow{Root}, x)$ (Figure 3.10).
3. Since v is on the path $(\overrightarrow{Root}, x) \in G_1$ and from the defintion of mutually linkable graphs, the path $(\overrightarrow{Root}, x) \in G_1$ is the same as the path $(\overrightarrow{Root}, x) \in G_2$, v must be on this path and thus must exist in G_2.
4. From our assumtion above however, v does not exist in G_2, therefore by contradiction x cannot be contained in G_2.

∎

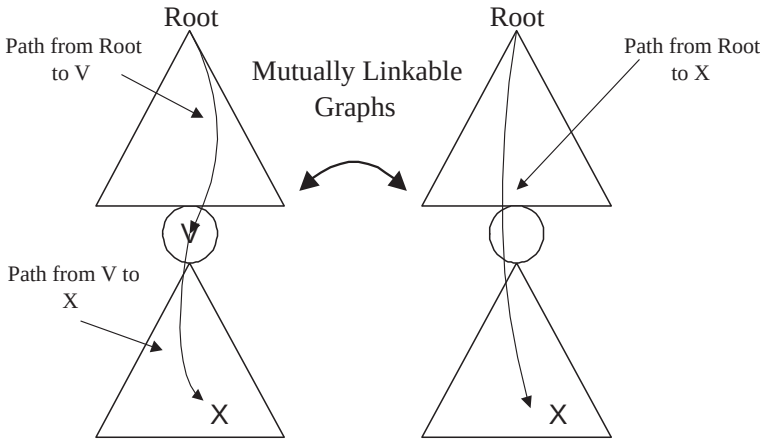

Figure 3.10: Linkable Graph Property

3.5.4 Theorem: Abstraction commutes with linking

In order to allow for individual units to be abstracted and then linked it is important that every abstraction operation $\mathcal{A}$, and the linking operation $\mathcal{L}$ commute.

$$\mathcal{A}(\mathcal{L}(G_1, G_2)) = \mathcal{L}(\mathcal{A}(G_1), \mathcal{A}(G_2))$$

Since linking is the union of two mutually linkable graphs, the requirement above can be written as:

$$\mathcal{A}(G_1 \cup G_2) = \mathcal{A}(G_1) \cup \mathcal{A}(G_2)$$

Proof: Since an abstraction is the removal a vertex, the removal of an edge or the addition of a new edge in the form of a lift, we have to show that all three of these operations commute with linking. The operation $e\mathcal{R}_{T(v_1,v_2)}(G)$ represents the removal of the directed typed edge $T(v_1, v_2)$ from graph G, $v\mathcal{R}_v$ represents the removal of the vertex v from from graph G, and $e\mathcal{D}_{T(v_1,v_2)}$ represents the addition of the directed typed edge $T(v_1, v_2)$ in graph G. Therefore we have to show that the following three properties hold:

1. $e\mathcal{R}_{T(v_1,v_2)}(G_1) \cup e\mathcal{R}_{T(v_1,v_2)}(G_2) = e\mathcal{R}_{T(v_1,v_2)}(G_1 \cup G_2)$
2. $v\mathcal{R}_v(G_1) \cup v\mathcal{R}_v(G_2) = v\mathcal{R}_v(G_1 \cup G_2)$
3. $e\mathcal{D}_{T(v_1,v_2)}(G_1) \cup e\mathcal{D}_{T(v_1,v_2)}(G_2) = e\mathcal{D}_{T(v_1,v_2)}(G_1 \cup G_2)$

Property 1: $e\mathcal{R}_{T(v_1,v_2)}(G_1) \cup e\mathcal{R}_{T(v_1,v_2)}(G_2) = e\mathcal{R}_{T(v_1,v_2)}(G_1 \cup G_2)$

Since the union operation does not introduce any new edges, if an edge is removed from two mutually linkable graphs, the edge will not be present in the union of the two graphs.

Property 2: $v\mathcal{R}_v(G_1) \cup v\mathcal{R}_v(G_2) = v\mathcal{R}_v(G_1 \cup G_2)$

Since the union operation does not introduce any new vertices, if a vertex is removed from two mutually linkable graphs, the vertex will not be present in the union of the two graphs.

Property 3: $e\mathcal{D}_{T(v_1,v_2)}(G_1) \cup e\mathcal{D}_{T(v_1,v_2)}(G_2) = e\mathcal{D}_{T(v_1,v_2)}(G_1 \cup G_2)$

In order to demonstrate property 3, we have to show that if the deduced edge exists in the union of the two mutually linkable graphs, it exists entirely within at least one of the two graphs.

From the definition of linking in section 3.5.2, two mutually linkable graphs are composed of a set of vertices V some of which present in both graphs ($G_1 \cap G_2$), known as shared vertices and some of which are only present in only one of the two graphs, known as non-shared vertices. From the property 3.5.3 for any non-shared vertex v, all vertices on the path $(v, \overrightarrow{leaf})$ are also non-shared ($\forall\ leaves \in G$). This means that once a non-shared vertex is reached on a path from the the root to the leaves, all the remaining

vertices on that path are also non-shared. This means that G_1 and G_2 can be drawn as Figure 3.11.

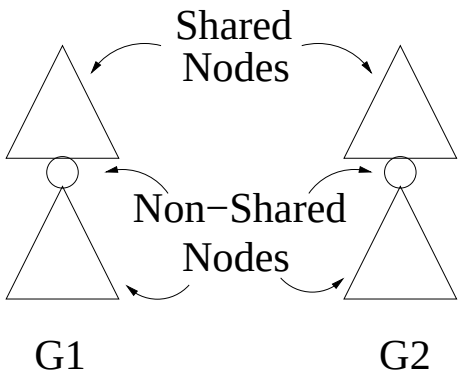

Figure 3.11: Graph Structure

C* $\circ$ u $\circ$ P* can be re-written as $(\overrightarrow{w,x}) \circ (x,y) \circ (\overleftarrow{y,z})$ where $(\overrightarrow{w,x})$ is S* (a path of structural edges), (x,y) is ns (a non-structural edge) and $(\overleftarrow{y,z})$ is $\tilde{\text{S}}^*$ (an inverted path of structural edges). If we can show that the composition $(\overrightarrow{w,z})$ is entirely contained within one of G_1 or G_2, then we have shown that property 3 holds.

S* and $\tilde{\text{S}}^*$ will always be contained in one of G_1 or G_2 since $(\overrightarrow{w,x})$ is a sub-path of $(\overrightarrow{Root,x})$ and $(\overleftarrow{y,z})$ is sub-path of $(y,\overleftarrow{Root})$ It just remains to be shown that ns (which is (x,y)) cannot originate in G_1 and terminate in G_2. There are 4 possibilities for ns.

1. (x,y): x and y both exist in shared space. If this is the case then the entire composition $(\overrightarrow{w,z})$ lie in shared space (Figure 3.12).

2. (x,y): x and y both lie in non-shared space. Since the two non-shared spaces cannot have any edges between them, x and y must lie in the same non-shared space and thus the composition $(\overrightarrow{w,z})$ exists entirely in that graph (Figure 3.13).

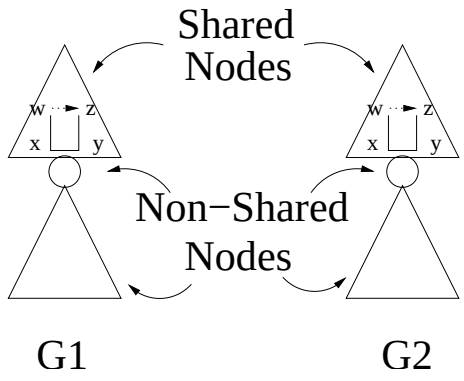

Figure 3.12: First Possibility

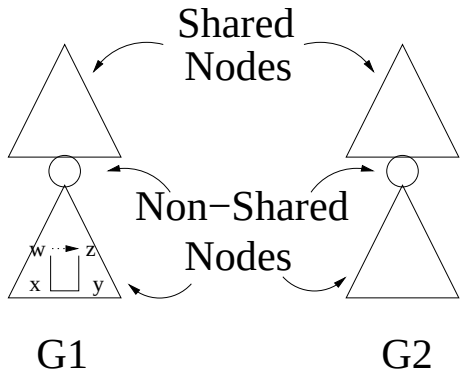

Figure 3.13: Second Possibility

3. (x, y): x is in shared space, y is not. In this case we choose the graph that y lies in. x must also exist in this graph since it is in shared space. Therefore the edge (x, y) is in the graph, thus $(\overrightarrow{w, z})$ lies in the graph (Figure 3.14).

4. (x, y): x is in non-shared space, y is in shared space. From the same argument presented above, (x, y) must be in one graph (Figure 3.15), thus $(\overrightarrow{w, z})$ is contained entirely within that graph.

Therefore since we can compose (w, z) entirely within a single graph, the deduced edge is deducible in the single graph and therefore we have shown property 3 holds.

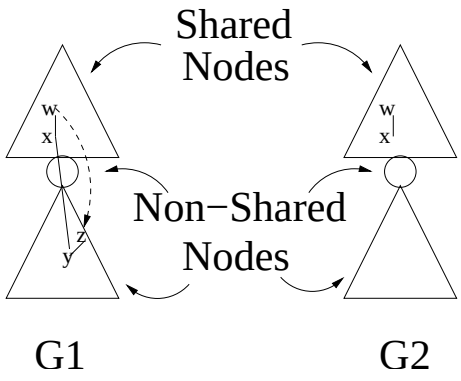

Figure 3.14: Third Possibility

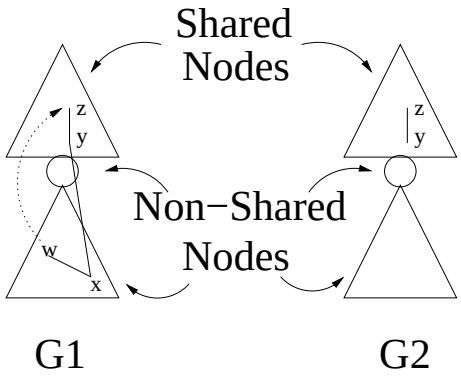

Figure 3.15: Forth Possibility

Since we have shown that all three properties hold, we have shown *abstraction commutes with linking*. ∎

3.6 Summary of Abstraction for Reverse Engineering

While studying a piece software, one of the most challenging aspects is determining what can be left out of the study. While determining what can be left out, it is also important to figure out what new information must be deduced. We call the process of leaving out of consideration some information and deducing new knowledge from the information left

out, *abstraction*. Abstraction is a fundamental step in reverse engineering and is provided in one way or another by every reverse-engineer tool. Tools which are used specifically for one task, such as producing class diagrams, must leave out of consideration everything not associated with class diagrams. Reverse-engineering tools of boarder scope that give the user all the information about a software system, must allow for abstraction or else the user will be presented with too much data and find it impossible to navigate.

All of the current tools used for reverse engineering have an abstraction phase either explicitly or implicitly. Some perform the abstractions while the facts are being extracted, whereas others provide a toolkit and let the user do the abstractions themselves. The abstractions performed fall into one of two categories, 1) Data Abstractions or 2) Schema Abstractions. Schema abstractions are abstractions specified on the schema and automatically applied to the data. Data abstractions are abstractions that cannot be specified on the schema and must be directly applied to the data. Through the use of schema abstractions new schemas can be deduced to limit the view of the system. Some views include the “Type View”, “Call Graph” and “Class Interaction View”.

Since many systems are too large to analyze and even too large to abstract, the abstractions must be split and the results linked together. We have formally defined abstractions and linking and shown that these two operations commute. Because of this criteria, not all filtering operations performed by the reverse engineering tools can be considered an abstraction. Abstractions can now be categorized as a pattern language.

CHAPTER 4

Data Abstraction Patterns

All in all it's just another brick in the wall.

– Roger Waters, The Wall (1979)

As chapter 3 explains, abstractions can provide valuable aid in the reverse engineering process by filtering the extraneous information from a model representing a software system and provide a better representation of the model for program understanding. Abstractions include the removal of vertices, removal of edges and lifting of edges. The next two chapters present a list of abstractions that meet this definition. The list was compiled by looking at the reverse engineering tools studied in section 3.2. Each abstraction in the list has been specified as a pattern, describing the core of the solution for a problem the abstraction will assist with.

This chapter presents **Data Abstractions**, abstractions which are applied directly to

the data. Chapter 5 extends this chapter by presenting the **Schema Abstractions**. This chapter starts by looking at what an abstraction pattern is (Section 4.1) by outline the properties of the pattern language. In section 4.2 each of the five data abstraction patterns are explained in detail.

4.1 What is an Abstraction Pattern?

An abstraction pattern is a high level description of an abstraction. Each pattern describes a common problem that occurs during the reverse-engineering process and then presents the core of the solution to the problem.

Patterns are not algorithms, they are descriptions of a solution. As Christopher Alexander said (and Gamma et. al Design Patterns Book [GHJV95] reiterated) *"each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* Alternative methods of applying each pattern are presented along with the resulting consequences. A pattern cannot be implemented as a single algorithm to handle all cases, but rather the pattern specification will allow the software engineer to develop his own specific solution for the environment he is working in.

This thesis splits abstraction patterns into two sections, *Data Patterns* and *Schema Patterns*. Of course each pattern will be eventually applied to a factbase, schema patterns are patterns specified at a high enough level to actually have an effect on the underlying schema for the data model. Data patterns on the other hand will not uniformly affect the schema (Section 3.3.1).

Each abstraction pattern is demonstrated on a small example. The patterns are most useful when combined together. Since two abstractions are also considered an abstraction, any number of patterns can be applied together to form a very complicated abstraction. It is important to note however that these patterns do not necessarily commute with one another so the order in which they are applied is important.

4.1.1 Describing an Abstraction Pattern

Abstraction patterns have the following parts:

1. The **pattern name** is used to identify the pattern. We have tried to use names descriptive enough that someone familiar with the patterns will be able to quickly understand what it is the pattern does.

2. The **intent** section outlines what will be achieved through the application of this pattern. A common problem that can be addressed through the use of the pattern is presented.

3. The **motivation** section gives a more detailed description of the type of problem that can be addressed by the pattern. This section outlines why these situations can present a problem for design recovery and shows how the application of the pattern can solve the problem.

4. The **applicability** section gives concrete examples of when this pattern should be applied and what will be achieved through the application of the pattern in each case.

5. The **consequences** section presents things that should be considered when applying the pattern. These may include side effects or possible alternatives to consider when applying the pattern.

6. The **known uses** section outlines a list of known uses for this pattern and some details of the systems which employ them.

7. The **related patterns** section describe other patterns that are related to the pattern or are often used in conjunction with the pattern to produce a more complicated abstraction.

4.2 Data Patterns

Data patterns are applied to the factbase. After the pattern is applied a refocusing operation is used to remove all vertices in the fact base no longer reachable from a root node. Some patterns use edge lifting to deduce new non-structural edges. Since edge lifting meets the defenition of a lift in section 3.5.1, these patterns meet the definition of abstraction and thus commute with linking.

Most of the patterns to follow will use the example shown in Figure 4.1 to show the effect the pattern would have on the graph representation of a small compiler.

The rectangles in this graph represents subsystems, the circles represent functions and the dashed lines represents structural edges. Each function is contained in exactly one subsystem. The solid edges represents “call” edges.

The small compiler example in figure 4.1 consists of 7 subsystems: *driver, scanner, parser, semantic, optimizer, codegen* and *library*. There are 52 functions in this example

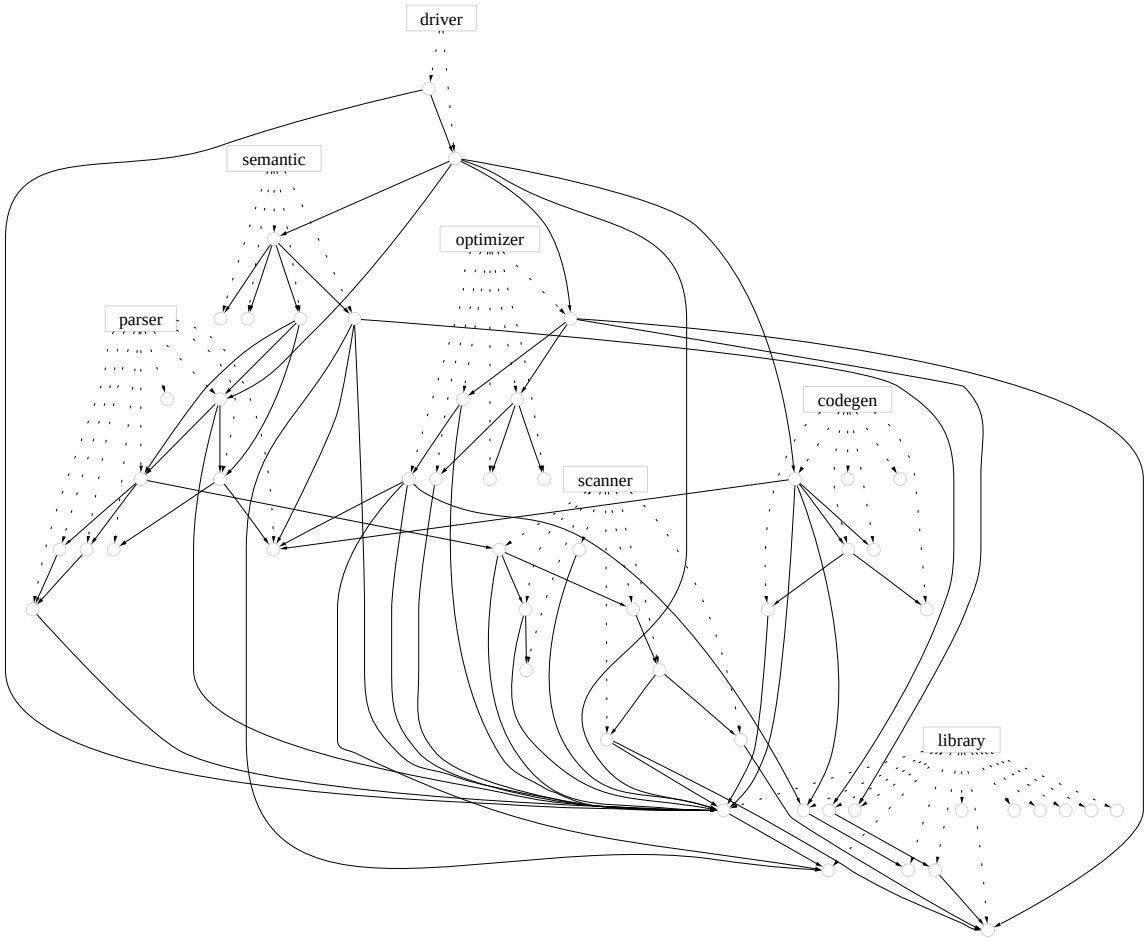

Figure 4.1: Compiler Example

and 67 call edges.

4.2.1 Subtree Removal

Intent

Remove the clutter that standard library routines and other high traffic entities produce.

Motivation

Certain entities clutter the model because either every entity interacts with them (as the case is with libraries) or they interact with every other entity (as the case is with drivers). It is often desirable to hide these entities. In the example shown in Figure 4.1 it is hard to understand how the subsystems interact with one another because of the large number of call edges cluttering the diagram. Many of these call edges reference a library routine. There appears to be one function in the library that most other routines depend on. After inspection, it might be determined that this function is a console write utility and can be removed from the diagram because it helps us to understand the system.

A software engineer can use the subtree removal pattern to remove a subtree by selecting the root node of the subtree and removing it. This is because all vertices must be reachable through a path of structural edges and if any vertex is removed all descendants of that vertex will not be reachable. Figure 4.2 shows the diagram after the subtree rooted at the “Library” has been removed.

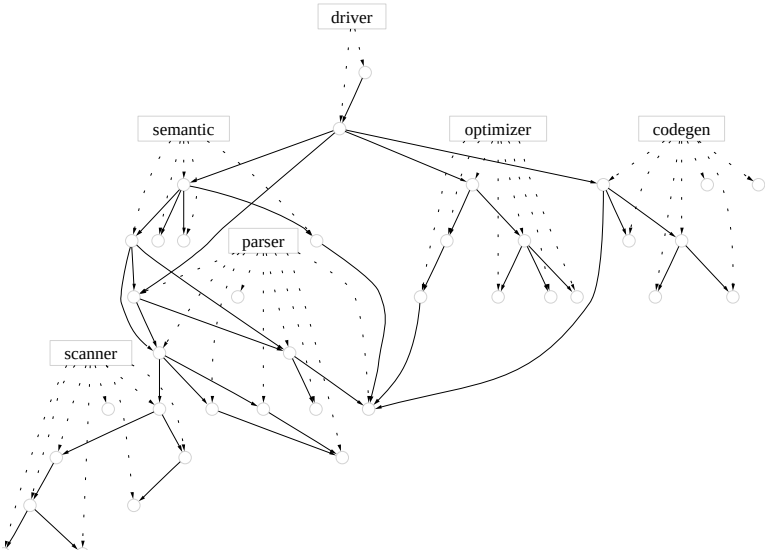

Figure 4.2: Library Subsystem Removed

Applicability

Subtree Removal is useful whenever there are nodes in the data model which add clutter without displaying much information. This pattern can be applied to help remove:

- Extraneous **subsystems** in architectural models such as libraries.
- Utility **classes** in class diagrams which all other classes depend on.
- Driver **functions** in call graphs who interact with all other functions.

Consequences

The following should be considered when applying the Subtree Removal pattern:

1. *All relations dependent on this entity are removed.* If this is applied to a library routine, all references to this routine are removed. If a query is then performed to

count the number of function calls that exist in the system, all calls to the library routine will not be included in the count.

2. *No trace of the entity is left*. No trace of the deleted vertex, or tree rooted at this vertex, will be left. It may produce deceptive diagrams if poorly used. Often a line of prose accompanying the diagram explaining what was done will help. The Foreign Edge Removal pattern (82) also provides a way of restricting the interaction with a high traffic node without actually removing the node.

3. *Selecting the nodes to remove*. The nodes to remove can be selected either manually by hand or automatically through a threshold of relations. For example, if more than 90% of all the nodes in the system are related to the node it can be removed.

Known Uses

This pattern is available in almost all tools in some form or another. Dali requires the software engineer to write a relational query to determine which relations should be removed. The Swagkit has filter which will remove tree nodes by specifying the root vertex. A very interesting implementation of this pattern is available in Rigi:

1. *Remove Omni-present Nodes*. Rigi supports the functionality to remove Omni-present Nodes.

> *This function is a parameterized operation designed to filter the noise at the initial stages of the subsystem composition process. For each node* $v \epsilon V$ *is a resource-flow graph* $G = (V, E)$*, let* $c(v)$ *be the number*

> *of direct clients of v. If $c(v)$ is greater than the omnipresent threshold T_{op}, the v is said to be omnipresent. Because omnipresent components obscure system structure, they are often removed from an RFG together with all their incident edges (by cutting or filtering). An example of an omnipresent node is a debugging module containing debug variables or routines that are referenced by most other modules. [MOTU93]*

Related Patterns

The following patterns are related to the Subtree Removal pattern:

- Subtree Elision (73). Elide an entire subtree to a single node leaving only the root node. All dependency edges connected to any node contained within the subtree are directed to the specified node.
- Foreign Edge Removal (82). Remove all edges that are not fully contained in a set of nodes to help focus on a single subtree.
- Edge Removal (99). Remove an edge from the schema, and remove the all instances of that edge from the data.

4.2.2 Subtree Elision

Intent

Elide an entire subtree to a single node leaving only the root node. All dependency edges connected to any node contained within the subtree are directed to the specified node.

Motivation

Removing large amounts of extraneous data from libraries and other components unrelated to program comprehension is a common task provided by many reverse engineering applications. In the Subtree Removal pattern (69), the entities and relations are removed, leaving no trace of the subsystem at all. In some cases it may be desired to keep the edges but remove the vertices. This is a common activity performed on function bodies and library subsystems. By performing this abstraction, all relations are re-directed to the root node and all other vertices are removed.

Figure 4.3 shows how the library subsystem in compiler example from Figure 4.1 can be elided. This unifies the entire subtree into the library node and all relations to and from any node contained within this subsystem are now directed at the library node itself.

Applicability

Data Subtree Elide is often used to hide the contents of a subtree whose structure is uninteresting or unimportant for the task at hand. Libraries or subsystems unrelated to

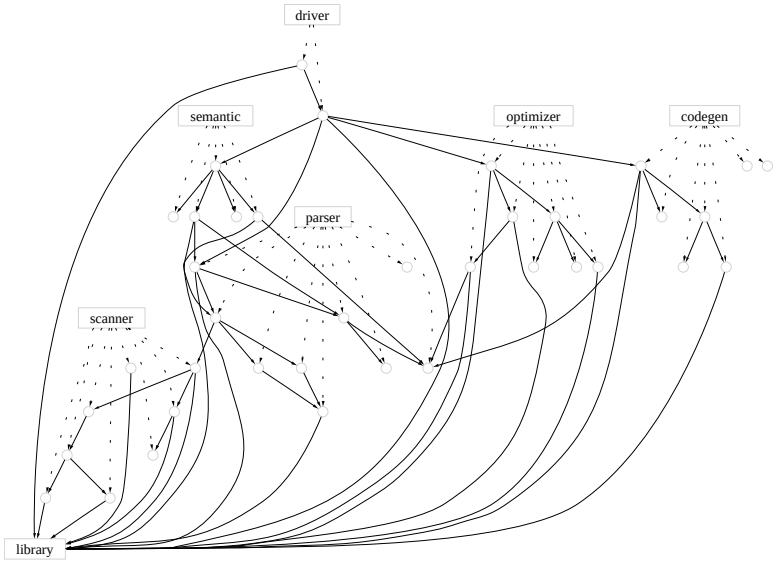

Figure 4.3: Library Subtree Elision

design recovery task are good targets to apply this pattern to. This pattern, unlike the Subtree Removal pattern (69), maintains relations between the subtree and other subsystems. This pattern is ideal when a software engineer wishes to study interconnections between subsystems, but not the details of the subsystems.

If it is the intent of the software engineer to elide to all vertices of a given type, for example elide all the “Subsystems”, then the Entity Lifting pattern (103) should be considered.

Consequences

Here are some issues to consider when using the Subtree Elision pattern:

1. *What happens to vertices structurally contained below the elision point?* The vertices of the tree can either be completely removed from the database, returned to the user for further processing or simply hidden.

2. *How do internal calls get lifted?* If there is a call between vertices contained within the subtree (or a self loop on a single vertex within a subtree) a self loop may be created on the elided vertex. If this is done, Figure 4.3 would be extended to include a self loop on the library subsystem.

3. *Is the same edge repeated?* If a vertex x has an edge to two vertices a and b, located in the same subtree. If that subtree was elided to its root r the number of deduced edges to r could either be 1 or 2. Either both edges can be kept, causing a repeated edge (x, r) or the repeated edge can be combined. Most graph representations will automatically elide automatic edges resulting in a loss of information. The number of edges each deduced edge represents can be attached to the edge as an attribute.

4. *Sibling Subtree Elision.* In order to study the way subtree's interact, this pattern can be applied to all vertices that are considered the "siblings" of the vertex in question. For example if someone was interested in studying the "optimizer" subsystem in our compiler example, they could see what methods within this subsystem interact with other subsystems by eliding all other subsystems. This can be seen in figure 4.4

5. *What happens to attributes?* The attributes of elided nodes may be important in understanding the system. For example, if there was at attribute attached to a statement that indicated that the statement may divide by zero, then it is probably important to lift that attribute to the function level indicating that the function may divide by zero. Attributes such as "filename" should probably be combined for all vertices being lifted to the package level and attributes such as "line number"

should probably be returned as a range.

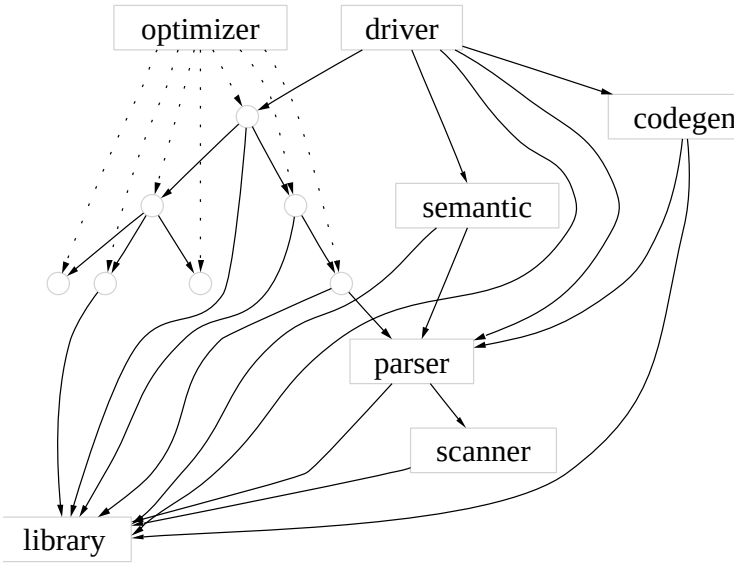

Figure 4.4: Elision of Siblings

Known Uses

The idea of elision is a commonly used in many of the reverse engineering tools. PBS, Rigi and Dali each allow for data specific elision:

1. *PBS / Swagkit* supports Subtree Elision through `lsedit's` open/close container feature. When a container is opened the edges point directly to the vertices they are attached to. When the container is closed, the vertices are hidden and the edges are elided to the root node. If multiple edges are attached to the same vertex, then when the subsystem is elided only a single edge is drawn to represent the group of edges.

2. *Dali's* abstractions are performed through relational queries. This particular abstraction is known as aggregation. Dali provides this abstraction by marking up the data contained within the tree with a special prefix and then that prefix is specified to mean the whole tree.

3. *Rigi* uses the idea of collapsing to combine a selection of nodes together. In the case of Rigi, it is not required that the nodes be structurally related, however, if it is going to commute with linking then they will need to be.

Related Patterns

The following patterns are related to the Subtree Elision pattern:

- Subtree Removal (69) Remove the clutter that standard library routines and other high traffic entities produce.

- K Depth Elision (78) Limit the data model to at most K levels and elide all relations below the K^{th} level to that level.

- Edge Lifting (107) Lift some contents of a container to the root without lifting all of it.

- Entity Lifting (103) Lift all the descendants of a entity specified in the schema.

4.2.3 K Depth Elision

Intent

Limit the data model to at most K levels and elide all relations below the Kth level to that level.

Motivation

Although browsing architectural diagrams rarely involves traversing more than a few levels, the information at the depths of the graphs determins the high level architectural edges. K Depth Elision traverses a tree K levels deep and then elides of all the relations that exist below this level. All the information found below these nodes is "lifted" to these nodes. This allows the information to propagate upwards and reside at the levels actually being studied by the architect. Figure 4.5 is typical of large software systems. Rarely do the high level entities have relations between them other than the ones deduced from the lower level entities because most of the interesting relations in a program occur within functions, or statements. Figure 4.6 shows what happens when the pattern is applied with $K = 2$.

Applicability

K Depth Elision is ideal for very large systems in which the software architect is trying to uncover the architectural relationships. Often a system consisting of a few thousand compilation units will not link unless the data is lifted to a high enough level. K Depth

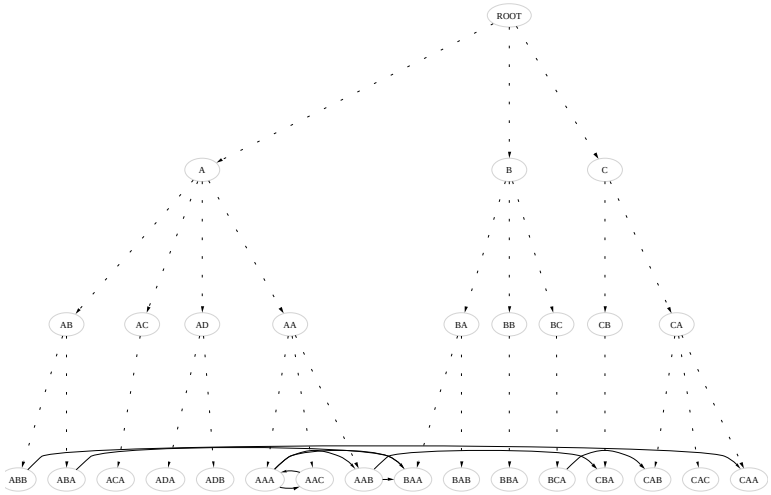

Figure 4.5: 4 Level Diagram

Elision works best if the containment is added to each compilation unit and then each unit can be abstracted individually. If the data is still too large to link it can be abstracted further. Nodes residing below the threshold will be lost, however all the edges will be "lifted". K Depth Elision, unlike Entity Lifting (103), elides to the K^{th} level regardless the entities type. This is useful when a graph has varying levels of depth as in the case of an unbalanced system [BFG+00].

This pattern also allows the Software Engineer to split the graph across levels. If a graph is too large to fit into memory, the graph could be split into k level chunks. Initially the first k levels could be loaded. Once a user selects a level below k, that graph could be unloaded and the next k levels could be loaded.

Consequences

The following should be considered when using the K Depth Elision pattern:

1. *Each compilation unit should contain all necessary subsystem containment struc-*

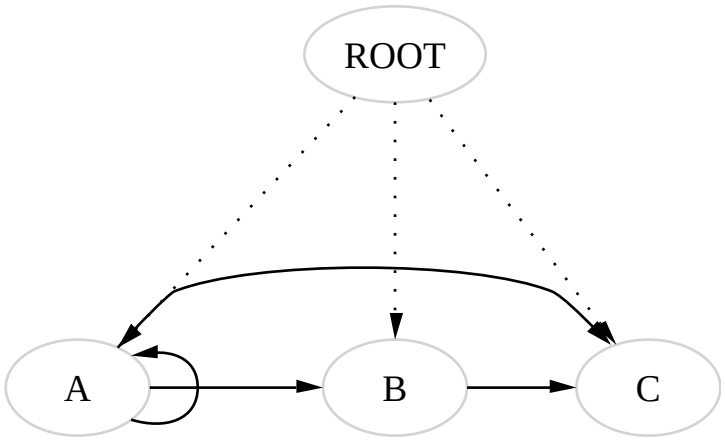

Figure 4.6: Lifted to Level 2

ture. Often in architectural analysis, the subsystem containment is not determined until after the entire system is linked together. However, if the abstraction is going to lift entities to the subsystem level, it is important that this information be present before the abstraction takes place. Each compilation unit should be augmented with its subsystem decomposition structure so the abstraction can take place before the linking.

2. *General elision problems.* Like the Subtree Elision pattern (73) and any of the patterns that perform elision on subtrees, questions such: *as should self loops be drawn, what happens to the removed nodes* and *how are attributes lifted*, must be addressed. See the consequences on page 74 for more information on these.

Known Uses

We have found one use of this pattern among the tools we studied:

1. *PBS.* The original Portable Bookshelf software included this abstraction in the

landscape viewer because of limited amounts of memory. Only a small number of levels could be loaded at a time and once a user traversed below that number of levels the graph would change.

Related Patterns

The following patterns are related to the K Depth Elision pattern:

- Subtree Elision (73). Elide an entire subtree to a single node leaving only the root node. All dependency edges connected to any node contained within the subtree are directed to the specified node.

4.2.4 Foreign Edge Removal

Intent

Remove all edges that are not fully contained in a set of nodes to help focus on a single subtree.

Motivation

While attempting to recover the design of a single subsystem, it may be desirable to remove all edges affecting the subsystem from external sources. For example, the developer who is trying to update a library package may not wish to see who is using the library, he may only wish to see the internal structure of the library. Figure 4.7 shows how this pattern can be applied to our example to remove all the relations which cross subsystem boundaries.

In the context of a compiler, someone who is trying to better understand the parser, may not wish to see how the parser is being used, only how it works internally. The pattern can also use the direction of the edge when determining if the edge should be kept or not. For example, if the edge originates within a particular set of nodes, then the edge can be kept otherwise it could be discarded. Figure 4.8 shows the compiler example in which only edges that originate within the parser subsystem are drawn. Figure 4.9 extends this idea even further and shows all the edges (x, y) such that either x or y is located within the parser subsystem.

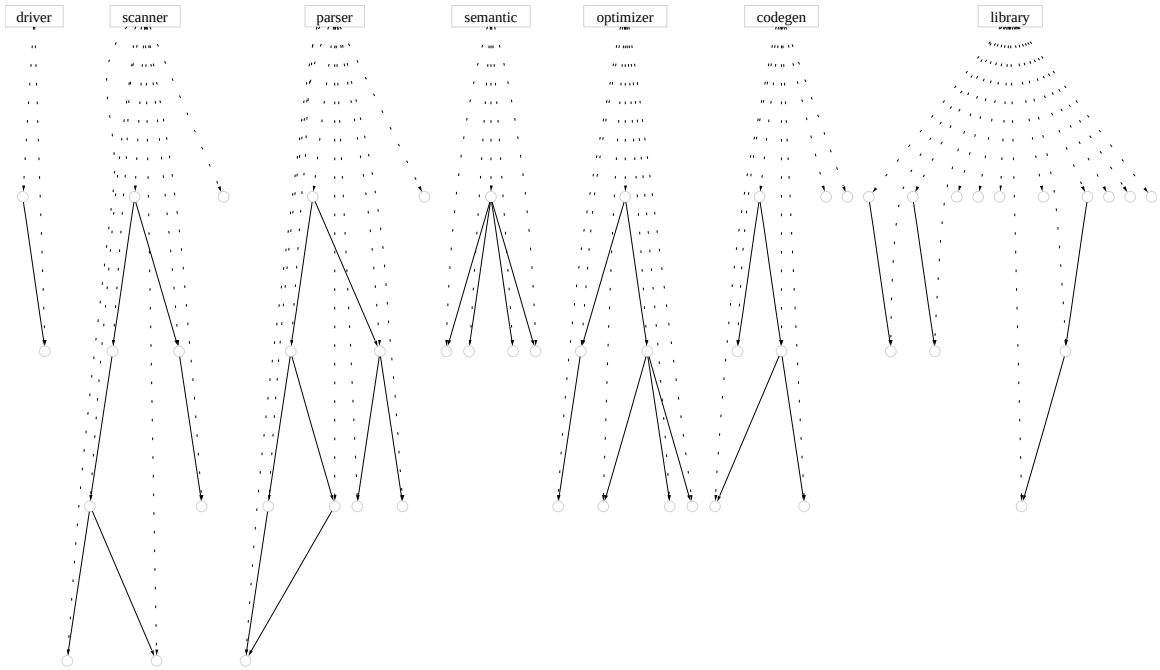

Figure 4.7: Foreign Edge Removal

Applicability

This pattern is used to help address two common issues which arise during reverse engineering:

1. *Subsystem Focused Analysis.* Once the reverse engineering activity requires the software engineer to begin focusing on a small set of entities, this pattern can help remove edges which are not of interest.

2. *Interface Analysis.* The Foreign Edge Removal pattern can help a software engineer get an idea of how a class is being used or what other classes are considered dependencies for this class. In order to see this, the pattern will have to be applied with edge direction in mind. For example, if the pattern is applied such that any

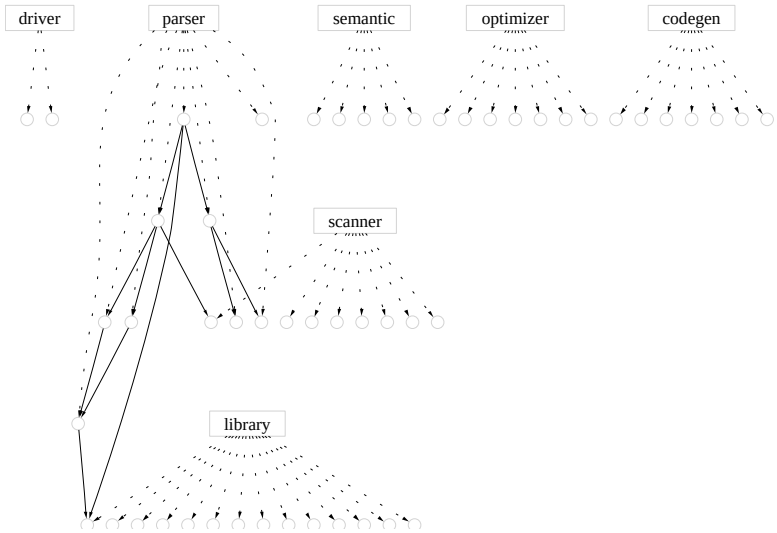

Figure 4.8: Parser as a Supplier

edge which originates within a particular class is kept, it will be easy to see exactly how the class works both internally and how it uses other classes. Of course, if those classes have dependencies of their own, they will not be represented.

Consequences

The following should be considered when applying the *Foreign Edge Removal* pattern:

1. *Edge direction interpretation.* In some cases you may wish to see which subsystems the one in question is depended on, but not which subsystems are dependent on it. In other cases it may be desirable to remove all interactions with other parts of the system. In the first case, you should remove edges whose domain is in a foreign subsystem. In the second case all edges whose domain and range is not limited to the same subsystem will be removed.

2. *How are nodes specified?* The nodes are easily specified by giving a root to a

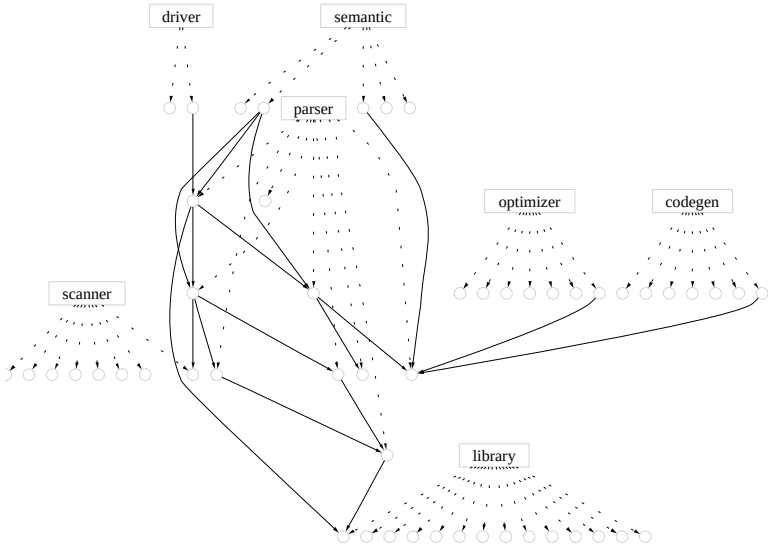

Figure 4.9: All Parser Interaction

subtree, however it is possible, and possibly useful to allow a user to specify the list of nodes as a set. This way, even if the containment is not correct, particular relations, which are not fully contained within a subtree can be kept.

Known Uses

Any tools which support nested views, such as SHriMP [WS00], Ciao and `lsedit`, will usually support this pattern. SHriMP calls this the *Fish-Eye View*, while Ciao calls this a *Focused* view.

Related Patterns

The following patterns are related to the Foreign Edge Removal pattern:

- Domestic Entity Removal (87) Removes all entities from a subtree that are only referenced from other entities within that subtree, or not referenced at all.

- Subtree Removal (69) Remove the clutter that standard library routines and other high traffic entities produce.

4.2.5 Domestic Entity Removal

Intent

Removes all entities from a subtree that are only referenced from other entities within that subtree, or not referenced at all.

Motivation

Standard libraries such as the Standard IO library for C, STL for C++ and the Java Libraries are filled with methods, data structures and macros. A single application will only use a small percentage of these routines and many of these routines are not even exported, they are only used internally. When analyzing a system it may be desirable to know what library routines are called, but the ones that are not used can be removed to help reduce the size of the model.

In our small compiler example only 50% of the library routines are actually referenced. Figure 4.10 shows the result of this abstraction applied to the Library subsystem.

Applicability

There are two primary applications of this pattern:

1. *Interface Analysis.* The Foreign Edge Removal pattern (82) shows the dependencies for a given subsystem of a software system. This pattern, when applied on a subsystem, will help show who depends on this system and what methods are used as the interface.

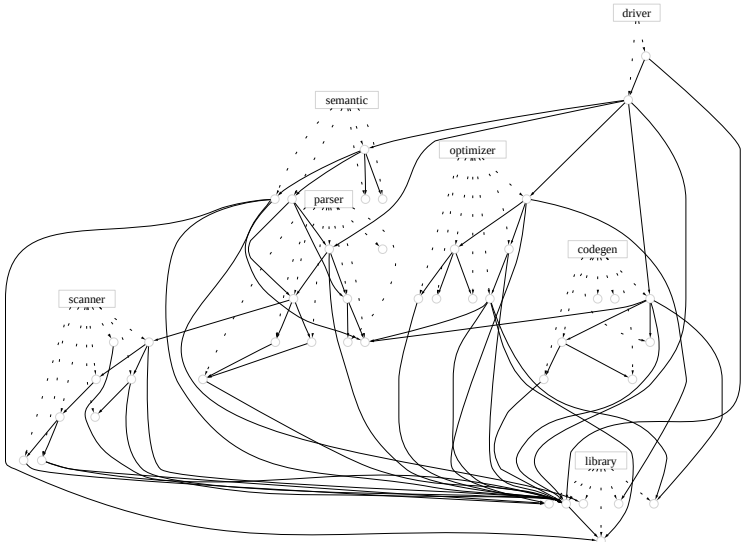

Figure 4.10: Domestic Entity Removal Applied to the Library Subsystem

2. *Filtering Unused Routines.* Removing unused libraries can often decrease the size of the data model considerably. When applied to the entire system, all routines that are never used will be removed. When applied to a subset of the system, all the routines which are not referenced from within that subset will be removed.

Consequences

The following should be considered when applying the Domestic Entity Removal pattern:

1. Unused routines are sometimes of interested to software engineers especially if they are attempting to uncover dead code.

Known Uses

Swagkit uses this pattern after linking to remove unused library routines.

Related Patterns

The following patterns are related to the Domestic Entity Removal pattern:

- Foreign Edge Removal (82). Remove all edges that are not fully contained in a set of nodes to help focus on a single subtree.

4.3 Summary of Data Abstraction Patterns

The patterns described in this chapter, Data Abstraction Patterns, are abstractions that are applied directly to the data. Five of these patterns have been recognized and each applied to a small example. Data Abstraction Patterns have no uniform effect for removing vertices or edges from the schema and are usually applied when trying to better understand part of the system or remove information from part of the system.

CHAPTER 5

Schema Abstraction Patterns

The significant problems we face cannot be solved at the same level

of thinking we were at when we created them.

– Albert Einstein (1879-1955)

CHAPTER 4 introduced the notion of Abstraction Patterns and explained what constitutes an Abstraction Pattern. The chapter defined five patterns of abstractions which can be directly applied to the data. This chapter will continue where chapter 4 left off by looking at **Schema Abstractions**, abstractions which can be applied to the schema. Section 5.1 explains what a schema abstraction is and then presents six patterns. Section 5.2 describes some properties which schemas should adhere to, to give a software engineer maximal flexibility when applying the patterns and to insure that the post abstraction filtering operation produces a schema which represents the data. Section 5.3

shows how multiple patterns can be combined and then applied to a system in sequence to present some very powerful views. The chapter then concludes with a summary of Schema Abstraction Patterns (Section 5.4).

5.1 Schema Patterns

As section 2.3.3 explains, schemas define the types of entities and relations which may exist in a factbase. Schemas also define what types of entities can be related to one another through what relations. We further constrained our data models by categorizing relations as either structural or non-structural, and entities are either root or non-root. We then required that each factbase be a forest with respect to its structural edges, rooted at a root node.

A schema abstraction is applied to the schema and induces a corresponding abstraction on the data. After a schema abstraction pattern is applied, a refocusing operations is performed. This insures that all schema vertices are reachable through a path of structural edges from a root node. If any vertex in the schema is not reachable from a root vertex then any instance of that entity in the data will not be structurally reachable and therefore the schema vertex will be removed. The standard refocusing operation that was described in Section 4.2 is then applied to the data removing all unreachable data vertices.

In the case of the compiler which we have used in many of our examples thus far, the schema was relatively simple since the entities in the model are only functions and subsystems. For the examples in this chapter we will use use the schema in figure 5.1.

Double octagons are considered to be root entities and rounded vertices are non-root

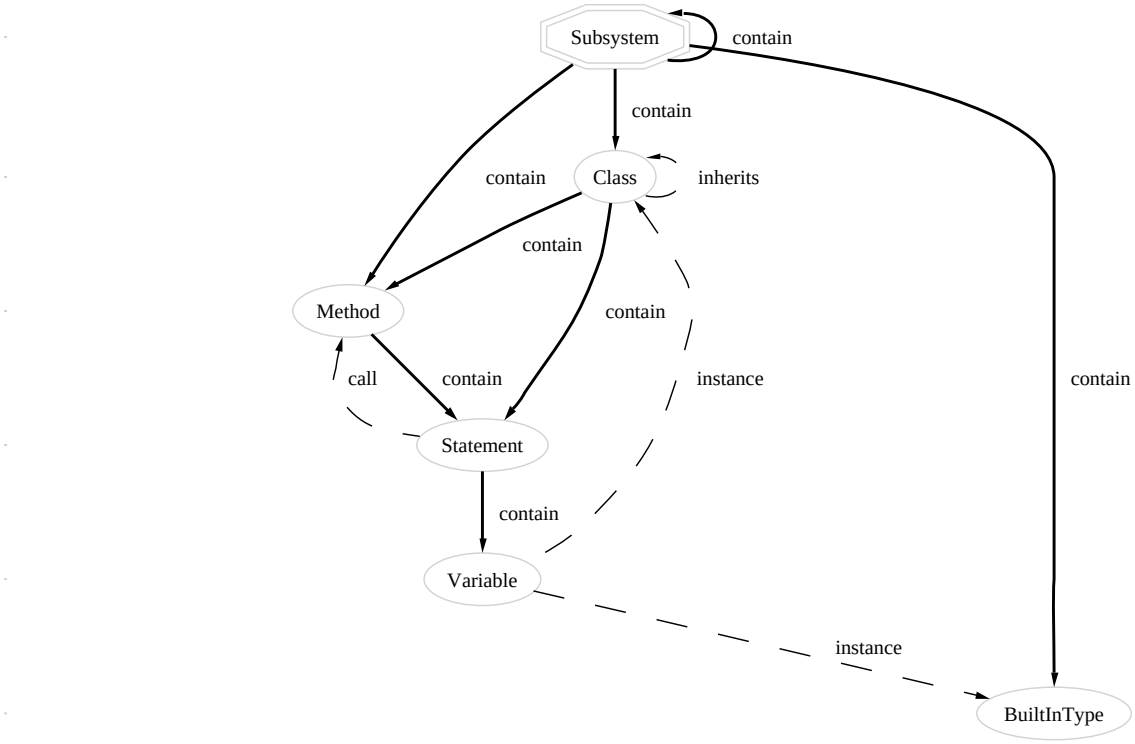

Figure 5.1: Schema

vertices. The solid arrows are the structural edges and the dashed arrows edges are the non-structural edges.

The Schema Abstraction Patterns are described using the same organization as was used for the Data Abstraction Patterns described in chapter 4, with the addition of the Data Application Section. The data application section describes how the pattern should be applied to a factbase after it has been applied to the schema.

5.1.1 Entity Removal

Intent

Remove an entity from the schema and all instances of that entity from the data.

Motivation

When studying call graphs and other flow control models in non object oriented languages, type and data information is not needed and can be removed to help simplify the model. If the abstraction is applied directly to the the data, the software engineer will have to find all the variables and type information in the system and then remove them. By specifying the abstraction on the schema, the software engineer can remove "Types" and "Variables" vertices and insure that all instances of them in the data will be removed. Since every entity in the schema must be reachable from a root entity through a path of structural edges, removing a single entity can easily result in the removal of a large amounts of data. In the example schema (Figure 5.1), if a software engineer was interested in studying the "Methods" and "Classes" which exist in a system, entities such as "Statements" and "Variables" could be removed. Furthermore, since all variables must be structurally contained in a statement, removing the "Statement" vertex will implicitly remove the "Variable" entity as well.

Figure 5.3 shows what would happen to our example schema if the "Statement" entity was removed. From this example, not only was the "Statement" entity removed but the "Variable" entity as well. This is because it does not make sense to keep a variable

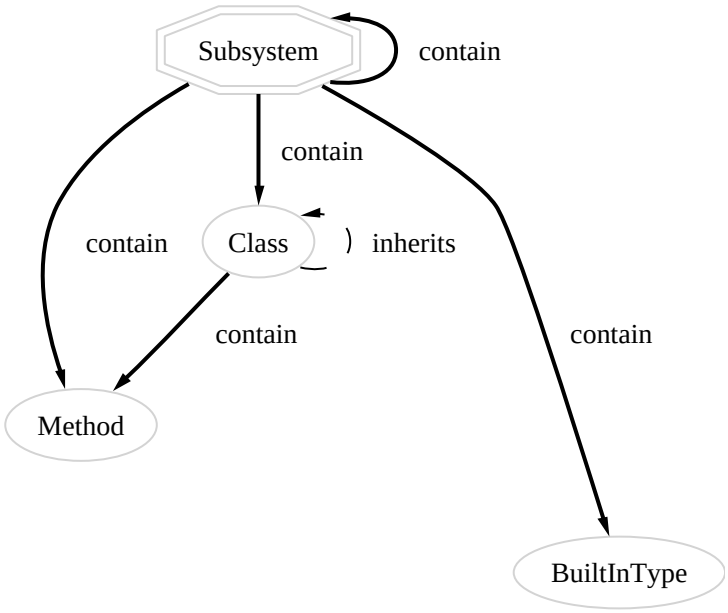

Figure 5.2: Entity Removal Applied to the Statement Entity

without the statement it is located in. Also, any edges which are related to Statements or Variables (such as the “Call” relation from “Statement” to “Method”) are also removed. If a software engineer wants to retain these relations then the Edge Lifting (107) or Entity Lifting (103) pattern should be considered. Furthermore, if a software engineer wishes to remove a single node from the schema without removing the nodes that are structurally contained within that node, then the Entity Bypass pattern (115) should be explored.

Applicability

This pattern is useful for removing large amounts of data rooted at a particular node (including the removal of the selected node itself). If it is desirable to keep the selected vertex, removing all of its descendants, then all the structural edges rooted at this vertex should be removed. The Edge Removal pattern (99) can be used for this.

Consequences

The following should be considered when applying the Entity Removal pattern:

1. *All entities that cannot be reached are removed.* The name of this pattern may be slightly misleading since more than just the node specified may be removed. The pattern removes all the vertices that are no longer reachable. If the nodes in the fact base which are to be deleted have descendants, then the descendants will also be deleted.

2. *Specify new roots.* It may be valuable to specify new vertices as root vertices before the pattern is applied. If the "Variable" vertex was specified as a root vertex in our example, then it (and all of this descendants) would not be filtered during the refocusing operation.

3. *No new edges are deduced.* No edge deduction is performed when this pattern is applied. Therefore any edges which rely on a descendant of the deleted node will be lost and no deduced edge will replace them. Since many of the interesting relations in source code happen at the depths of the tree, it may be desirable to deduce some new edges before applying this pattern. The Entity Lifting pattern (103) and Edge Lifting patterns (107) may help with this.

4. *Similar to Subtree Removal.* This pattern is very similar to the Subtree Removal pattern (69) but instead of a single subtree being removed, all subtrees rooted at a particular node type will be removed.

Data Application

To apply this pattern on a fact base, a list of all vertices in the fact base which are instances of the entity that has been selected to be removed, will be deleted from the model. The standard data refocusing operation will then take place removing all vertices in the data no longer reachable from the root.

Known Uses

All tools which employ a query language such as Dali and most tools which use a graphical user interface provide an operation similar to this pattern.

1. *Query Language.* Tools such as Dali which support a query language will inevitably support this pattern by allowing user to project the types of data they wish to see.

2. *GUI Tools.* Tools such as Rigi Edit support this pattern by allowing users to filter out entities based on their type.

Related Patterns

The following patterns are related to the Entity Removal pattern:

- Subtree Removal (69). Remove the clutter that standard library routines and other high traffic entities produce.

- Edge Removal (99). Remove an edge from the schema, and remove the all instances of that edge from the data.

- Entity Lifting (103). Lift all the descendants of a entity specified in the schema.

5.1.2 Edge Removal

Intent

Remove an edge from the schema, and remove the all instances of that edge from the data.

Motivation

Often the same schema entity represents two or more different types of nodes in the fact base. In our schema (Figure 5.1) for example, the method node represents both methods contained within classes and global functions. If a software engineer were only interested in studying global functions and wished to ignore the methods contained within classes, he could not perform this task by removing a node from the schema. Instead the structural edge connecting “Class” and “Method” should be removed. Since class methods would no longer be reachable in the data, they, along with all their descendants (the statements and variables contained within them) would be removed. The effect of this pattern can be seen in Figure 5.3.

In this example, the method vertex remains because it is still reachable from the “Subsystem” vertex, however the edge between the “Class” and “Method” nodes has been removed.

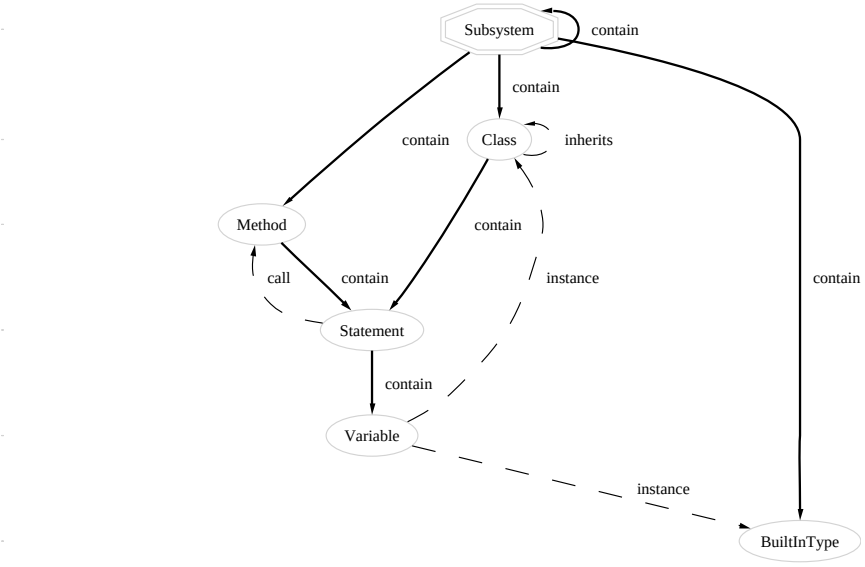

Figure 5.3: Edge Removal Applied to the Structural Edge From “Class” to “Statement”

Applicability

This pattern can be used to remove either a structural or non-structural edge from the schema.

- *Structural Edge.* If a structural edge is selected to be removed, then an entire subtree could potentially be removed if the edge was a structural bridge in the schema.

- *Non-Structural Edge.* If a non-structural edge is selected to be removed then no refocusing operation is necessary. This is because all data reachable before the abstraction was performed will still be reachable afterwards.

Consequences

The following should be considered when applying the Edge Removal pattern:

1. *Similar to subtree removal when applied to a structural edge.* This patterns works on both structural and non-structural edges, but since the pattern is followed by a refocusing activity, if it is applied on a structural edge it may leave nodes unreachable causing the filtering operation to remove these nodes.

2. *Can new roots be specified?* It may be desirable to allow for new root nodes. For example, if someone was only interested in "types" and "variables" they should both be specified as roots since there is no structural dependencies between these two elements.

3. *Has little effect on poor schemas.* Entities are only removed if they are no longer structurally reachable. If a schema is designed poorly and consists of extra structural edges, this pattern may not actually remove any entities since they may be reachable in the schema even if they are not reachable in the data. Section 5.2 expands upon this point.

Data Applications

To translate this to data operations every instance of the structural edge selected in the schema should be removed. The data refocusing operation should then be applied removing all entities which are not structural reachable from the root.

Known Uses

Many tools allow for the removal of edges through some sort of visual editor. Both `lsedit` and Rigi-Edit have facilities to allow the user to select an edge type and hide it

from view. However, none of the reverse engineering tools studied have the refocusing operation once a structural edge has been remove.

Related Patterns

The following patterns are related to the Edge Removal pattern:

- Subtree Removal (69). Remove the clutter that standard library routines and other high traffic entities produce.
- Entity Removal (94). Remove an entity from the schema and all instances of that entity from the data.
- Edge Lifting (107). Lift some contents of a container to the root without lifting all of it.

5.1.3 Entity Lifting

Intent

Lift all the descendants of a entity specified in the schema.

Motivation

In the first two schema abstractions, the Entity Removal (94) and Edge Removal (99) patterns, all relationships which relate entities in a removed subtree will be deleted themselves. Sometimes (for reasons described in the Subtree Elision pattern (73)) it is desirable to lift the edges to the root of the subtree before removing the vertices. If this pattern was applied to our example from Figure 5.1, by lifting the entire schema to the Subsystem level, all the relations would loop to and from the Subsystem as in Figure 5.4.

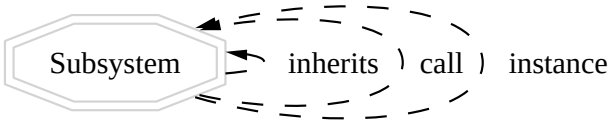

Figure 5.4: Subsystem Lift

To see how this would affect the factbase, when Entity Lifting is applied to our compiler example from chapter 4 (Figure 4.1) all the relations that exist below the subsystem level would be lifted to the subsystem, resulting in Figure 5.5.

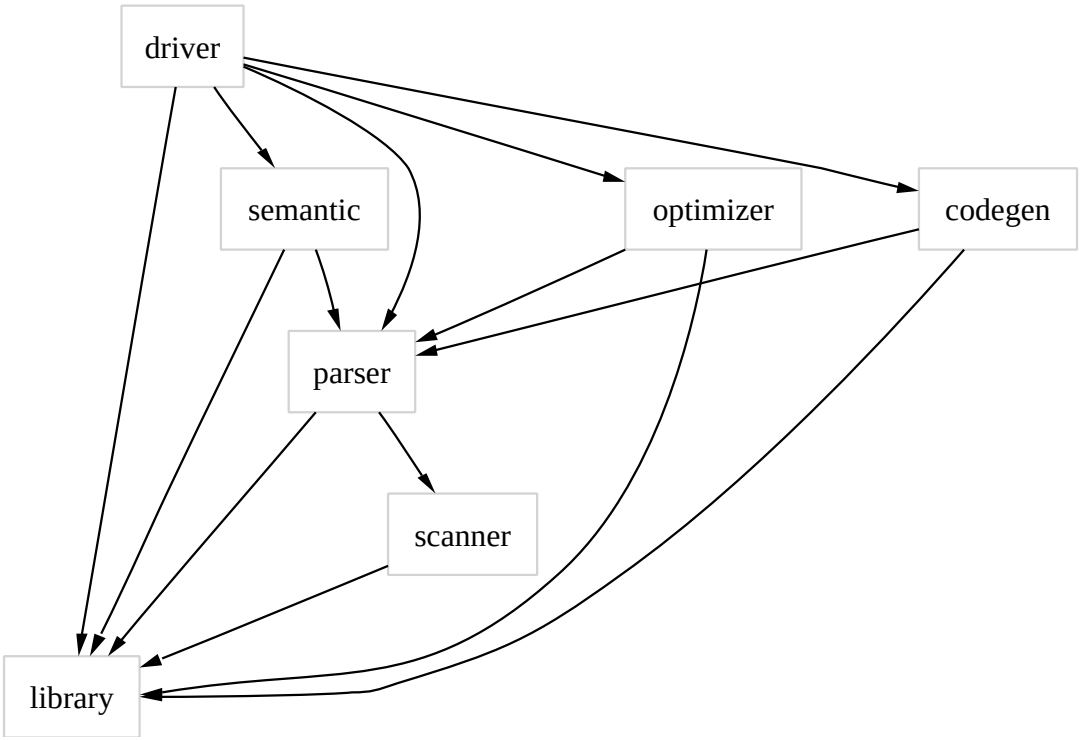

Figure 5.5: Compiler Subsystem Lift

Applicability

This pattern is used to elide all entities below a given entity type in the graph. "Functions", "Class" and "Subsystems" are common targets for this pattern. In fact, many of the different "Levels" schemas represent (AST Level, Middle Level, Architectural Level) choose a different set of nodes and apply this pattern (Section 2.4). For example, the middle level models will choose entities such as "Functions" and "Classes" and then lift to them. High level models will do the same thing to entities such as Module or Subsystem.

Consequences

The following should be considered when applying the Entity Lifting pattern:

- *Self Structural Loops.* Vertices such as “Subsystem” in our schema have self structural loops attached to them. This means that “Subsystems” are allowed to contain other “Subsystems”. When this pattern is applied it is often applied to the first entity that matches the specified type. This of course can be modified and patterns such as the K Depth Elision pattern (78) can aid with this.

- *General Elision Problems.* When this pattern is translated into data operations, the consequences of elision that were mentioned in the Subtree Elision pattern (73) on page 74 such as *presence of self loops, what happens to the removed nodes?* and *attribute lifting*, will have to be addressed.

- *Schemas should not contain unrealistic edges.* If entities are unreachable from a root vertex in the schema then they are unreachable in the data. Often, published schema’s have edges that don’t actually occur in the data. This will reduce the effect of schema operations such as this one. This is because nodes that are unreachable in the data are reachable in the schema and thus the schema vertices are not removed.

Data Application

This pattern is applied to the fact base by first selecting each entity in the fact base which is an instance of the selected schema node. The Subtree Elision pattern (73) is then applied to each entity in this set.

Known Uses

This is used in PBS and Swagkit as the standard lift operation. Bril et al. have also formalized this notion [BFG+00].

Related Patterns

The following patterns are related to the Entity Lifting pattern:

- Subtree Elision (73). Elide an entire subtree to a single node leaving only the root node. All dependency edges connected to any node contained within the subtree are directed to the specified node.
- Entity Removal (94). Remove an entity from the schema and all instances of that entity from the data.
- Edge Lifting (107). Lift some contents of a container to the root without lifting all of it.
- Entity Bypass (115). Remove an entity but maintain all children by directly attaching the children to their grandparents.

5.1.4 Edge Lifting

Intent

Lift some contents of a container to the root without lifting all of it.

Motivation

As the Edge Removal pattern (99) suggests, a single entity in the schema may represent two or more different types of nodes in the data. For example, the statement vertex may represent statements within function bodies or initalization statements within classes. If a software engineer wished to delete statements within functions, but not class statements, removing all statements will not help. The software engineer must lift along the structural edge that connects functions to their statements. Figure 5.6 shows what happens when the structural edge (Method, Statement) is lifted.

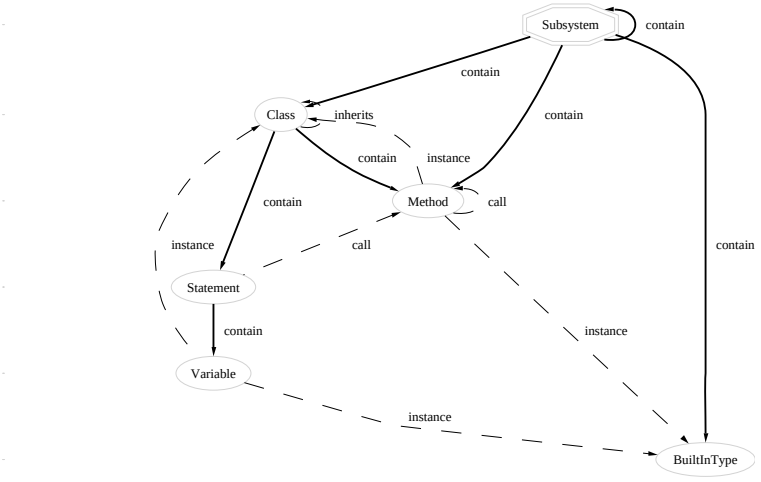

Figure 5.6: Edge Lifting

As with the Edge Removal pattern (99), statements are still present in the graph, but only the statements contained directly within classes. All the non-structural edges which relate statement or any descendants of statement are lifted to the Method vertex in the schema. The "call" is now a self loop on the "Method" vertex. This is the same as the Subtree Elision pattern (73) when applied to the fact base except only some of the children are lifted.

Applicability

Despite the name, this pattern can only be applied to a structural relation. It does not make sense to lift along a non-structural edge, and furthermore it would violate the definition of a deduction in an abstraction (Section 3.5).

There are two applications of this pattern that are useful:

- *Lift only some of the children.* If a software engineer only wishes to lift some of children of a node, for example, only lifting the global statements found in a class but not the routines, this pattern should be applied. If an entity is lifted instead of an edge, then all the children of "Class" will be lifted.

- *Lift certain paths to entities.* If a software engineer wishes to filter methods, which are routines found within classes, but not global functions, then this pattern should be applied.

Consequences

The following should be considered when applying the Edge Lifting pattern:

1. *Cyclic Schema's.* If the schema contains a cycle then lifting through the cycle will cause a lot of new edges to be deduced. this problem is expanded on in Section 5.2.

2. *General Elision Problems.* These are the same problems mentioned in the consequences for the Entity Lifting pattern (103) and expanded upon on page 74.

Data Application

In order to apply this pattern to a factbase, every vertex in the graph which is an instance of the domain of the selected structural edge selected is marked. Then the Subtree Elision pattern (73) is applied on this set, but instead of lifting all the children of this set, only the children that match the entity type of the range of the structural edge are lifted.

Known Uses

There are no known uses of this pattern.

Related Patterns

The following patterns are related to the Edge Lifting pattern:

- Entity Lifting (103). Lift all the descendants of a entity specified in the schema.

- Subtree Elision (73). Elide an entire subtree to a single node leaving only the root node. All dependency edges connected to any node contained within the subtree are directed to the specified node.

- Edge Removal (69). Remove the clutter that standard library routines and other high traffic entities produce.

- Entity Bypass (115). Remove an entity but maintain all children by directly attaching the children to their grandparents.

5.1.5 Entity Unification

Intent

Combines two or more entities together creating a single entity to represent all of them.

Motivation

Detailed schemas often create different entities that are conceptually very similar. For example, schemas will often differentiate between “Functions”, “Procedures” and “Methods”. While there are differences, for many activities it may be easier to think of all of them as “Routines”.

In the sample schema we have presented, user defined types “Classes” and “Built In Types” have been split into two separate nodes. A software engineer may wish to combine these together in order to study the variables and their data type without distinguishing between these two different data types. Figure 5.7 shows what happens when this is applied to our sample schema.

Applicability

This pattern is useful for combining entities which are conceptually similar, although they are represented as multiple entities in the schema because of slight semantic differences. In our experience this abstraction works to combine:

1. *Functions, Procedure, Methods*. These three distinct subroutine vertices can be combined into a single “Routine” vertex.

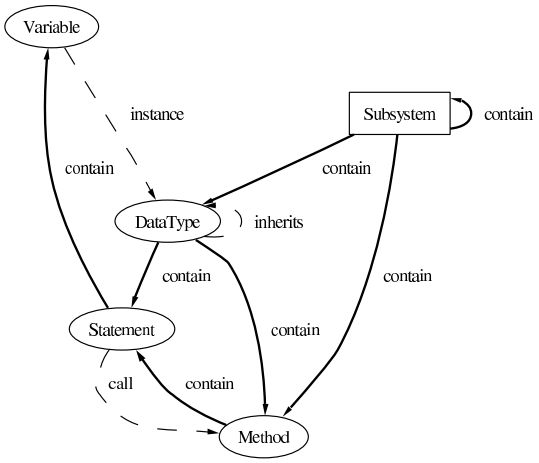

Figure 5.7: Type and Class Unification

2. *Types, Built In Types, Records, Classes.* These distinct type vertices can be combined into a single "DataType", "Sort", or "Classifier" vertex.

3. *While Statement, For Statement, Repeat Statement.* These three types of loop statements can be combined into a single "Loop" vertex.

If the vertices being combined are vastly different, combining them will produce a very complicated vertex. In the case of the different loop statements, each one may have a different type of loop counter. A combined vertex will allow relations to all three types of loop counters which may appear confusing. In order to help manage this complexity, it may be desirable to combine this pattern with the Edge Removal pattern (99) and remove the individual vertex specific edges.

Consequences

The following should be considered when applying the Entity Unification pattern:

1. *Unrealistic edges are produced.* In the example shown in Figure 5.7, “DataTypes” may contain methods. From the original schema however, “Built In Types” were not allowed to contain methods. This may appear confusing to someone studying the schema.

2. *Self Loops.* Any interaction between vertices which are combined will become self loops on the new unified vertex.

Data Application

In order to perform this operation on the data, every instance of the vertex in the set to be unified is modified to be an instance of the newly created unified node.

Known Uses

CIA++ (a C++ extractor) has only five entities in the schema (files, macros, types, variable and functions) [GC90]. Many of these entities are combinations of many other program entities. For example, variables could be local or global, functions could be global, static or methods and types could be user defined, built in, structures or classes. In order for CIA++ to give such a simple model for a complicated language such as C++ many vertices have been unified.

Related Patterns

The following patterns are related to the Entity Unification pattern:

1. Entity Removal (94). Remove an entity from the schema and all instances of that entity from the data.

2. Edge Removal (99). Remove an edge from the schema, and remove the all instances of that edge from the data.

5.1.6 Entity Bypass

Intent

Remove an entity but maintain all children by directly attaching the children to their grandparents.

Motivation

All the removal and elision abstractions that we have studied thus far remove large amounts of data by removing subtrees of unreachable vertices. It may be desirable to keep the vertices at the depths of the trees when removing some of their ancestors. Since every vertex must be reachable from a root node, there are two methods which can be employed to keep these low level nodes; 1) *Create new root nodes,* or 2) *Attach the nodes that would be removed to a pre-existing vertex that won't be removed*. This pattern explores the second option.

In the schema we have been studying up to now, a program understanding task may require variables, but not the statements which contain them. In this case the statement node could be removed and the variable node attached directly to the parent of the statement node. In other words, the statement node could be *bypassed*. Figure 5.8 demonstrates this pattern. Any relation which relates the bypassed node is composed with each structural edge into the node.

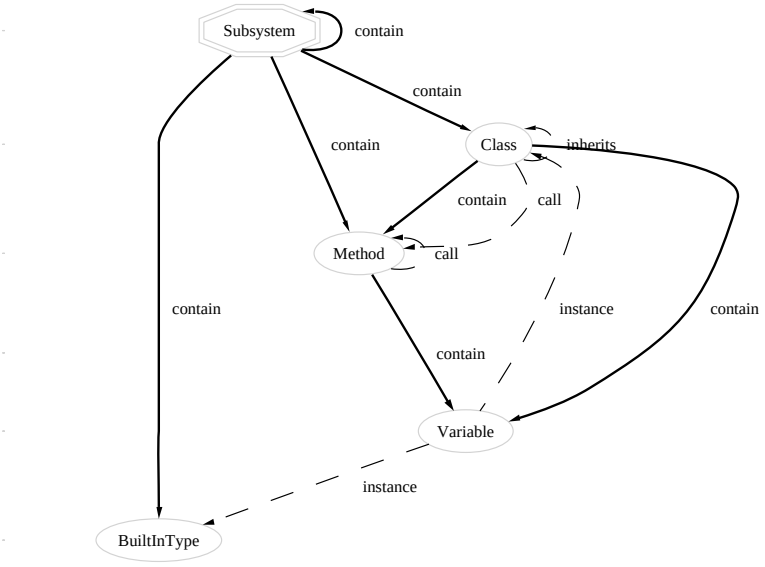

Figure 5.8: Entity Bypass

Applicability

This pattern is used to remove extraneous nodes from the schema without removing the descendants of such a node. Extraneous vertices are introduced for several reasons some of which are outline below:

- Grammars are defined with additional non-terminals as place holders when specifying lists. The grammar defined below demonstrates this:

  ```
  <function_body> --> <stmt_list>
  <stmt_list>     --> <stmt><stmt_list>
  <stmt>          --> ...
  ```

  A software engineer may consider a function body to consist of a list of statements, but it is often represented as a statement list, which in turn may contain a single

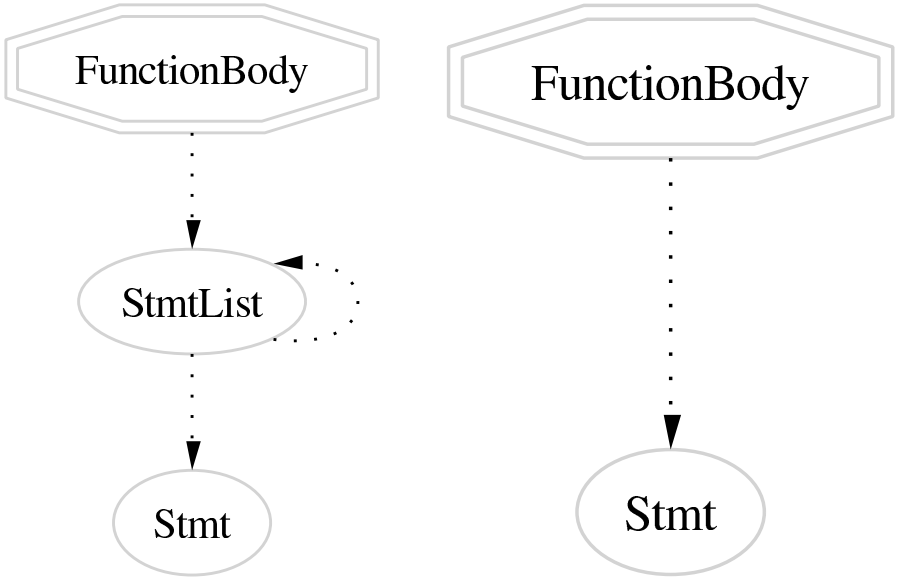

Figure 5.9: Effect of removing Statement List from the Schema

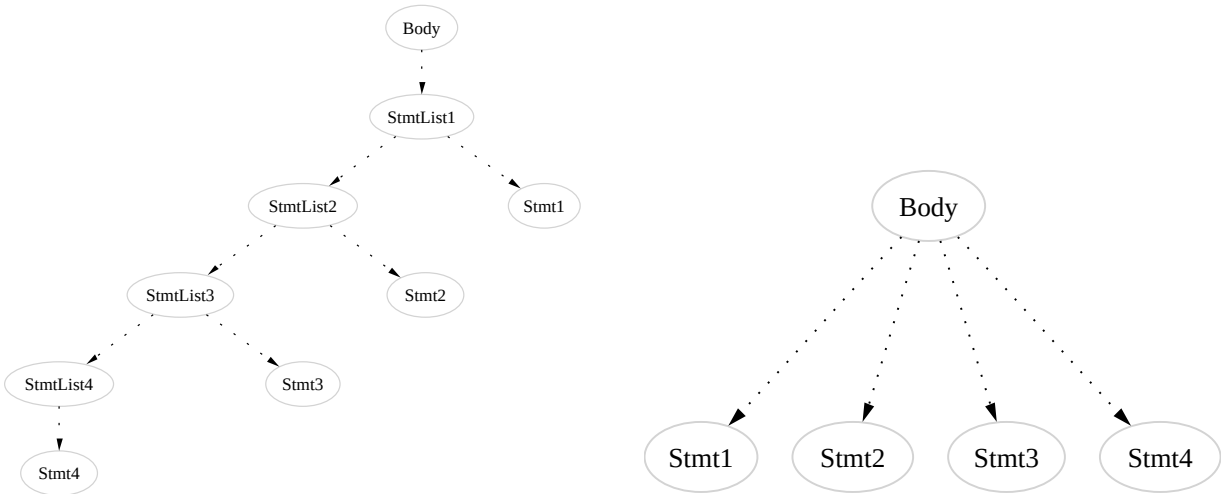

Figure 5.10: Effect of removing Statement List from a Factbase

statement and another statement list. To make program understanding easier, it may make sense to remove the "Statement List" vertex and just keep the statements. Each statement could then be a direct child of "Function Body". Figure 5.9 shows this transformation as it would affect the schema, and Figure 5.10 shows the factbase before and after the the transformation.

- *Structural Bridge.* If a schema vertex has only a single structural edge into and out of it, the node is considered a structural bridge. A node such as "FunctionBody" is an example of this. A function may contain a list of arguments, and a function body. The function body may contain a list of statements. In this case it may be desirable to remove the function body node and let the function contain a list of arguments and a list of statements. These additional vertices cluster entities nicely when they are being viewed, but add extra overhead if the data is meant to be queried.

Consequences

The following should be considered when applying the Entity Bypass pattern.

- *The entity will always be bypassed.* Both the Edge Removal (99) and Edge Lifting (107) pattern are used when the same vertex represents two conceptually similar entities. If a software engineer wishes to only bypass a vertex in certain circumstances (i.e. bypass methods, but not functions) then this pattern can be applied in a similar way as the other edges patterns. In this case the pattern would be known as an Edge Bypass or Edge Detour.

- *General Elision Problems.* Like the other patterns which employ the elision operation, problems such *presence of self loops, what happens to the removed nodes* and *attribute lifting* will have to be addressed. This are expanded upon page on 74.

Data Application

This pattern is applied to the data by taking all nodes which are instances of the selected vertex (y) and finding the parents of all these nodes (x). The structural edge (x, y) between the parent and child node is then composed with all edges (y, z) (edges rooted at (y)). These new edges (x, z) are then placed into the graph and the selected vertices (y) are then removed.

Known Uses

CPPX uses this pattern when it translates from the GCC representation of the AST to its own representation.

Related Patterns

The following patterns are related to Entity Bypass pattern:

1. Entity Lifting (103). Lift all the descendants of a entity specified in the schema.
2. Edge Lifting (107). Lift some contents of a container to the root without lifting all of it.

5.2 Schema Properties

Designing schemas for programming languages is a difficult task. The designer must decide how much detail to represent within the schema, what elements of the programming language should appear in the schema and what elements are allowed to be related. In order to insure the schema has maximal flexibility when it comes to abstraction patterns we have uncovered the following four properties that should be considered while specifying schemas.

1. *Do not include schema edges which will never occur in the fact base.* In order to use schema patterns effectively, the software engineer should be able to learn a lot about the data by looking at the schema. If the schema is littered with edge that cannot appear in the data, understanding the data through the schema will be a futile task. If, for example, the schema contained a structural edges from Variable to File a software engineer attempting to apply elision patterns will become confused very quickly. Each edge (both structural and non-structural) in the schema should have the possibility of existing in a fact base.

2. *Try not to combine similar vertices into one vertex.* Vertices such as Function and Methods should be kept separate in the schema. The Entity Unification pattern (111) showed a very easy way to combine these two entities, but illustrated the difficulties involved with splitting them once they have been combined. Because of this, the two entities should remain separate until the software engineer studying the data model wishes to combine them.

3. *Reduce the number of multi-node cycles*.

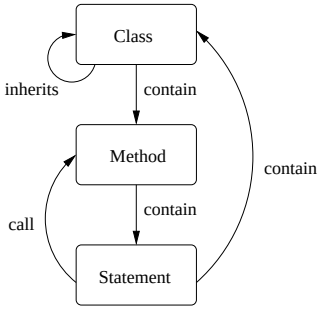

Figure 5.11: Schema with Cycle

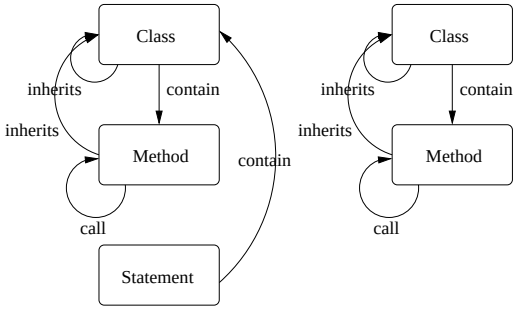

Figure 5.12: Edge Lifting Transformations

If the schema contains a cycle, such as the one in Figure 5.11, the edge lifting becomes complicated. If, for example, the "contain" relation (method,statement) was lifted (using the Edge Lifting pattern (107)) then the transformations in Figure 5.12 would take place. This has the strange effect of creating the relation inherit (method,class), but this should be read as "Something contained within a method can inherit from a class". This statement is true based on the schema presented since classes are contained within methods in the original schema. It is important to make sure that the original schema is correct or else the number of undesired relations could quickly increase.

4. *Choose root nodes carefully*. Since every node reachable from a root node will be not be removed during an abstraction filtering operation, it is important to select the root nodes with care. If too many vertices are marked as root then the abstraction operations will have little effect on the system.

5.3 Composition of Patterns

Applying a single abstraction pattern is usually not sufficient to in helping to understand a large system. A software engineer will usually apply multiple patterns, trying different alternatives as he learns more about the system. In the compiler example shown in Chapter 4 (Figure 4.1), if a software engineer was interested in how the driver subsystem interacts with the other subsystems, no single pattern would help. Figure 5.13 shows this interaction. This required the application of 3 patterns. First the library subsystem was removed (Subtree Removal pattern (69), then all the entities were elided to the subsys-

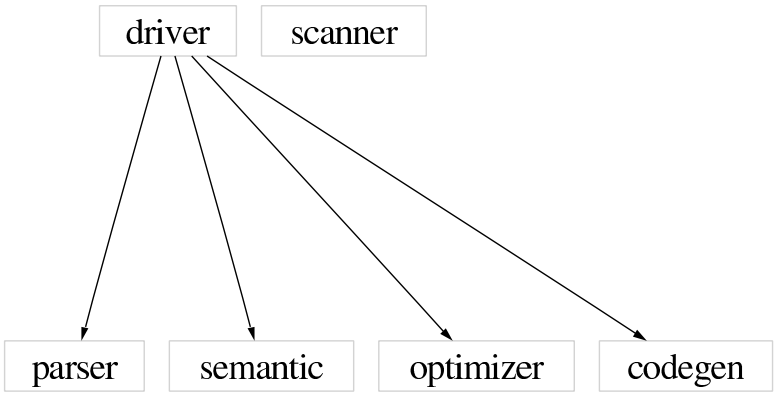

Figure 5.13: Pattern Composition

tem level (Entity Lifting pattern (103), then the Foreign Edge removal pattern (82) was applied keeping only the edges which originate from the “Driver”.

The true power of the patterns comes from the fact that they can be composed and still commute with linking. This means that a number of patterns can be applied to shape each individual compilation unit before the link stage takes place.

5.4 Summary of Schema Abstraction Patterns

Schema Abstraction Patterns are patterns of abstraction that are applied directly to the schema, resulting in a set of deducible operations to perform on a fact base. This chapter extended the notion of abstraction patterns by describing six schema patterns. These patterns were described the same way as the Data Abstraction Patterns shown in Chapter 4 with one exception, the data application section. This section described how to translate each pattern into a set of operations that could be performed on a factbase.

After the patterns were described, the chapter went on to explain some properties schemas should have, in order to insure that the schemas are well behaved during an

abstraction operation. The chapter concluded by looking at how multiple patterns can be composed to produce some very powerful views for a complicated fact base.

CHAPTER 6

Conclusion

No matter how good you are at something, there's always
about a million people better than you.
– Homer Simpson

In this thesis we looked at reverse engineering tools to determine if patterns exist in the manner in which the tools filter large factbases for further consumption, both by humans and other tools. We found that every reverse-engineering tool performs abstractions, the removal of detailed information and deduction of high level information, and we were able to categorize the abstractions performed into a collection of patterns.

6.1 Major Topics Addressed

In Chapter 3, four reverse engineering tools, PBS / Swagkit, Rigi, Dali and CIA / CIA++ / Ciao, were studied. Each of these tools support the reverse engineering process, which includes the extraction of source artifacts, the abstraction of source artifacts and the presentation of the artifacts. Each tool was studied to determine how it performs filtering operations on the its factbase. By looking at these filtering operations we were able to formally define the notion of *Abstraction* (Section 3.5). Abstraction was defined as *1) the removal of vertices, 2) the removal of edges* and *3) the deduction of new edges, where new high level edges are deduced from pre-existing low level edges.* We also emphasized the need for a tree-like structure, annotated with non-tree edges for semantic purposes.

From the four tools studied we determined that reverse engineering tools do indeed perform abstractions in this form at some point during the revere engineering process. It was also determined that some of the operations are performed by looking at the data, such as Rigi's removal of omni-present nodes, and other operations were performed by looking at the model which the data conforms to, such as PBS's lift to function. Because of the distinction between looking at the factbase and looking at the schema to which the factbase adheres, we were able to categorize a set of patterns into one of two categorizes: *1) Data Abstractions* and *2) Schema Abstractions*. By looking at these reverse engineering tools we were able to uncover five data patterns (Chapter 4) and six schema patterns (Chapter 5). Each of these patterns has been described in detail, along with a small example of how each could be applied in practice. Each abstraction is followed by a re-focusing operation which removes all nodes not reachable from the root of the tree. The abstraction, coupled with this refocusing operation results in a powerful tool

for a software engineer attempting to understand software. This is because a software engineer can properly filter out large amounts of data with minimal knowledge about the system.

6.2 Major Thesis Contributions

This thesis makes three main ontributions. First, the notion of abstraction was shown to commute with “linking”. This allows a software engineer to either abstract their data before they have linked their entire factbase, or after the linking stage, producing the same result either way. Because many large fact bases are too large to manipulate, this result allows a software engineer to manipulate parts of the model in parallel and link the abstracted graph afterwards. This result has allowed us to deal with fact bases larger than 1MLoc in size.

Secondly, this thesis has given names and descriptions to each of the patterns uncovered. Consequences and alternatives for each pattern were explained along with explanations of when each pattern can be applied. Such a pattern language can be used as a reference when building new reverse engineering tools which need to present large amount of data to a user.

Finally, this thesis showed how to translate from one schema to another through a set of abstraction patterns. This thesis also discussed the importance of removing the notion of levels from our schemas. Up to now all schemas that were considered to be “Low Level” contained high level information such as files, modules and subsystems. This thesis described the importance of schemas that contain only focused schema elements that may span different levels, but do not contain all higher level elements. For example,

the type / variable schema was proposed (Section 3.4) in order to show how types and variables interact in a large software system.

6.3 Future Work

This thesis has proposed several new research questions and concepts which should be integrated into current reverse engineering tools. This section outlines how this thesis can be extended and the ideas of this thesis integrated into other tools.

6.3.1 Software Reconstruction Through Abstraction

Since abstractions can be used to display large amounts of data in a easier-to-understand format, it would be ideal if a software engineer could use this "easier-to-understand format" to perform software reconstruction and forward engineering. They could then specify their requirements at a high level and a process would translate this into the low level operations. This would require a process opposite to abstraction in which low level operations could be deduced from high level ones.

6.3.2 Full Schema Transformation

If data can be easily migrated from one schema to another, then the need for standard schemas is no longer necessary. Each reverse engineering tool can use its own schema, and when data needs to be shared, a data transformation from one schema to the other could be performed.

Ideally, two schemas could be specified and transformations between the two could

be automatically deduced and specified as a collections of abstractions. Obviously not every schema translation is an abstraction so this would require additional research.

6.3.3 Integrated Reverse Engineering Environment

Even though abstractions are performed by each tool in the reverse engineering community, it is not easy for a novice user to get up to speed on process. If a reverse engineering environment could be developed which includes the abstraction patterns then a novice software engineer could specify his own views without extensive knowledge about the reverse engineering process. This will make it easier for the average developer to use the concept of reverse engineering for tasks such as program understanding, architectural reconstruction and re-engineering.

APPENDIX A

Basic Graph Theory Definitions

Definition A.1: *A **Directed Walk** in a graph $G = (V, R)$ is a finite sequence of vertices $v_o, v_1, ..., v_k$, such that (v_{i-1}, v_i), $1 \leq i \leq k$ is a relation in G.*

Definition A.2: *A **Directed Trail** is a directed walk in which all its edges are distinct. A directed trail is **open** if its end vertices are distinct; otherwise it is **closed**.*

Definition A.3: *A **Directed Circuit** is a closed directed trail in which all vertices except the end vertices are distinct.*

Definition A.4: *A directed graph G is **connected** if the underlying undirected graph is connected.*

Definition A.5: *A directed graph G is said to **acyclic** if it has no directed circuits.*

Definition A.6: *A directed graph is G a **tree** if the underlying undirected graph is a tree.*

Definition A.7: *A vertex v in a directed graph G is a **root** if there are directed paths from v to all the remaining vertices of G.*

Definition A.8: *A directed graph is G a **directed tree** or **arborescence** if G is a tree and has a root.*

Bibliography

[AT98] Matt Armstrong and Christopher Trudeau. Evaluating Architectural Extractors. In *Proceedings of WCRE'98*, Honolulu, Hawaii, October 1998. 11

[Bel] Bell Canada. http://www.bell.ca. 25

[BERP98] Susan K. Burkwald, Stephen G. Elick, Kurt D. Rivard, and John D. Pyrce. Visualizing Year 2000 Program Changes. In *Proc. Of IWPC*, pages 13–18, 1998. 10

[BFG+00] Reinder J. Bril, Loe M.G. Feijs, André Glas, René L. Krikhaar, and Thijs Winter. Hiding expressed using Relational Albegra with Multi-relations – oblique lifting and lowering for unbalanced systems –. In *Proceedigns of the Fourth European Conference on Software Maintenance and Reengineering*, pages 33–43, 2000. 79, 106

[BG99] Ivan T. Bowman and Michael W. Godfrey. Extracting source models from Java programs: Parse, disassemble, or profile? Available from `http://plg.uwaterloo.ca/ migod/papers/`, 1999. 14

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. of the* 21^{st} *Intl. Conference on Software Engineering (ICSE-21)*, Los Angeles, CA, May 1999. 4, 6, 10, 36, 37

[BI02] Larry A. Barowski and James H. Cross II. Extraction and Use of Class Dependecy Information for Java. In *Proceedings of the Working Coference on Reverse Engineering*, pages 309–315, 2002. 10, 15

[BMD+00] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontagiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of 7th Working Conference on Reverse Engineering*, pages 98–107, 2000. 11

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern - Oriented Software Architecture*. John Wiley & Sons, 1996. 7

[BR02] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: a Roadmap. In *Proc. Of IEEE International Conference on Software Maintenance*, 2002. 10

[BTMG02] R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey. Semantic Grep: Regular Expressions + Relational Abstraction. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 267–276, Oct. 2002. 10, 16, 37

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998. 11

[Car97] Luca Cardelli. Program Fragments, Linking, and Modularization. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, 1997. 33

[CC01] Anthony Cox and Charles Clarke. Representing and accessing extracted information. In *Proc. of 2001 International Conference on Software Maintenance, Florence, Italy*, November 2001. 10, 16

[CFKW95] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995. 36, 45, 46

[CGK97] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 414–431. Springer–Verlag, 1997. 47

[CH95] J. Cross and T. Hendrix. Using generlized markup and SGML for reverse engineering graphical representations of software. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 2–6, Toronto, 1995. 10, 16

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976. 16

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and deisgn recovery: a taxonomy. In *IEEE Software*, Volume: 7 Issue: 1, pages 13–17, Jan. 1990. 10, 11

[CNR90] Y. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, pages 3:325–334, 1990. 10, 14, 36, 45

[CSDM01] J. Cordy, K. Schneider, T. Dean, and A. Malton. HSML: Design directed source code hot spots. In *Proc. of International Workshop on Program Comprehension*, Toronto, Canada, May 2001. 10, 16

[DAG] Schloss Dagstuhl, International Conference and Research Center for Computer Science. http://www.dagstuhl.de/. 25

[Dal] The Dali Architecture Reconstruction Workbench. Website. http://www.sei.cmu.edu/ata/products_services/dali.html. 36, 42

[DIC] Webster's Revised Unabridged Dictionary Version published 1913 by the C. & G. Merriam Co. Springfield, Mass. Under the direction of Noah Porter, D.D., LL.D. 30

[FAB02] Rudolf Ferenc and Árpád Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of Sixth European Conference on Software Maintenance and Reengineering*, pages 59–66, 2002. 14

[FHK+97] Pat Finnigan, Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi Müller, John Mylopoulos, Steve Perelgut, Martin Stanley, and Kenny. Wong. The software bookshelf. *IBM Systems Journal*, 36(4), November 1997. 10, 36

[FKO98] L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software-Practice and Experience*, 28(4):371–400, 1998. 19, 38

[FSH+01] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimothy. Towards a Standard Schema for C/C++. In *Working Conference on Reverse Engineering*, pages 49–58, 2001. 24, 25

[GC90] Judith Grass and Yin-Farn Chen. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, April 1990. 14, 36, 45, 46, 113

[GCC] GCC: GNU Compiler Collection. http://gcc.gnu.org/. 14

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 7, 64

[GL00] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proc. of 2000 Intl. Symposium on Construct-*

ing software engineering tools (CoSET 2000), Limerick, Ireland, June 2000. 10, 36

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In *V. Ambriola and G. Tortora, editors, Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientfic Publishing Company. 25

[GXL] GXL The GXL Homepage. Website. http://www.gupro.de/GXL. 4, 14

[HH02] Ahmed E. Hassan and Richard C. Holt. Architecture Recovery of Web Applications. In *In Proceedings of 24th International Conference on Software Engineering*, pages 349 – 359, 2002. 10, 14

[HHL+00] Richard C. Holt, Ahmed E. Hassan, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Working Conference on Reverse Engineering*, pages 284–286, 2000. 14, 25

[Hol] GROK Richard C. Holt. The Grok Programming Language. http://plg.uwaterloo.ca/ holt/papers/grok-intro.html. 30, 36, 38

[Hol96] Richard C. Holt. Binary relational algebra applied to software architecture. Technical Report 345, University of Toronto CSRI, 1996. 18, 19

[Hol97] Richard C. Holt. An introduction to TA: The Tuple-Attribute language. Available at `http://plg.uwaterloo.ca/~holt/papers/ta.html`, 1997. 4, 14, 24

[Hol98] Richard C. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *Working Conference on Reverse Engineering*, pages 210–219, 1998. 15

[Hol99] R. Holt. Software Architecture Abstraction and Aggregation as Algebraic Manipulations. In *Proceedings of the IBM Centre of Advanced Studies Conference*, 1999. 15

[Hol02] Ric Holt. Software Architecture as a Shared Mental Model. In *Proc. of 2002 International Workship on Program Comprehension, Paris, France*, May 2002. 25

[HSW00] Ric Holt, Andy Schŭrr, and Andreas Winter. GXL: Towards a Standard Exchange Format. In *Proc. of 2000 Working Conference on Reverse Engineering (WCRE-00)*, pages 162–171, Brisbane, Australia, November 2000. 14, 23, 24

[IWP] IWPC International Worksip on Program comprehension. Website. http://www.iwpc2003.uvic.ca/. 10

[KBWA94] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994. 25

[KOV01] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture Reconstruction Guidelines. Technical report CMU/SEI-2001-TR-026, 2001. 42, 43, 44

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. 4, 26

[Kru95] Philippe B. Kruchten. The 4+1 View Model of Architecture. In *IEEE Software*, 12(6): 42-50, November 1995. 4, 5, 25

[KWC98] R. Kazman, S. G. Woods, and S. J. Carriere. Requirements for Integrating Software Architecture and Reengineering models: CORUM II. In *Proceedings of WCRE*, 1998. 11

[KWM] Holger M. Kienle, Anke Weber, and Hausi A. Müller. Leveraging SVG in the Rigi Reverse Engineering Tool. Website. http://www.svgopen.org/papers/2002/. 15, 39

[LA97] Timothy C. Lethbridge and Nicolas Anquetil. Architecture of Source Code Exploration Tools: A Software Engineering Case Study. Technical Report 97-07, University of Ottawa, Computer Science, 1997. 10

[Lee00] Eric Hon Sam Lee. Software Comprehension Across Levels of Abstraction. Master's thesis, University of Waterloo, 2000. 15, 22

[LIN] LINUX. Website. http://www.linux.org/. 5

[LPT+01] Timothy C. Lethbridge, Erhard Plödereder, Sander Tichelaar, Claudio Riva, and Panos Linos. The Dagstuhl Middle Model. 2001. 14

[LRY00] Raimondas Lencevicius, Alexander Ran, and Rahav Yairi. Third eye – specificiation-based analysis of software execution and traces. In *Proceedings of Internatinoal Conference on Software Engineering*, page 772, 2000. 14

[LS80] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addision Wesley, Readings, MA, 1980. 11

[MDH01] Andrew J. Malton, Tom R. Dean, and Richard C. Holt. Union Schemas as a Basis for a C++ Extractor. In *In proceedings of 8th Working Conference on Reverse Engineering*, pages 59–67, 2001. 2, 14, 30

[MJN+92] L. Miller, L. Johnson, J.Q. Ning, A. Quilici, and P. Devanbu. Program Understanding - Does It Offer Hope for Aging Software? In *In Proc. of 7th Knowledge-Based Software Engineering Conference*, pages 238 – 242, 1992. 34

[MK88] Hausi A. Müller and Karl Klashinsy. Rigi: A system for programming-in-the-large. In *Proc. of the* 10^{th} *Intl. Conference on Software Engineering (ICSE-10)*, Singapore, April 1988. 10, 36

[MK96] Nabor Mendonça and Jeff Kramer. Requirements for an Effective Architecture Recovery Framework. In *Proc. of the 2nd Int. Software Architecture Workshop*, pages 101–105, 1996. 11

[MMCG99] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of International Conference on Software Maintenance*, pages 50 – 59, 1999. 6, 12, 15, 30

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965. 32

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Software*

Maintenance: Research and Practice, pages 5(4):181–204, December 1993. 14, 72

[NK94] Stephen C. North and Eleftherios Koutsofios. Application of graph visualization. In *Proceedings of Graphics Interface '94*, pages 235–245, Banff, Alberta, Canada, 1994. 15

[oW] The University of Washington. Pine Information Center. http://www.washington.edu/pine/. 2

[PBS] PBS The Portable Bookshelf. Website. http://www.swag.uwaterloo.ca/pbs. 36

[Pen93] David Allan Penny. *The Software Landscape: A Visual Formalism for Programming-In-The-Large*. PhD thesis, University of Toronto, 1993. 15, 22, 37

[PSK96] J. Paukki, A. Salminen, and J. Koskinen. Automated hypertext support for software maintenance. *The Computer Journal*, 39(7):577–599, 1996. 16

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. In *ACM Sigsoft, Software Engineering Notes*, volume 17 no 4, pages 40–52, 1992. 5, 25

[RIG] RIGI The Rigi Group. Website. http://www.rigi.csc.uvic.ca/. 6, 39, 41

[RR02] Claudio Riva and Jordi Vidal Rodriquez. Combining Static and Dynamic Views for Architectural Reconstruction. In *Proceedings of Sixth European Conference on Software Maintenance and Reengineering*, pages 47 – 55, 2002. 14

[SCHC99] Susan Elliott Sim, Charles L.A. Clarke, Richard C. Holt, and Anthony M. Cox. Browsing and Searching Software Architectures. In *Proceedings of International Conference on Software Maintenance, Oxford, England*, 1999. 30

[SH98] Susan Elliott Sim and Richard C. Holt. The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. In *Proc. of the 20th International Conference on Software Engineering*, , Kyoto, Japan, 1998. 10

[SHE02] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On Using a Benchmark to Evaluate C++ Extractors. In *Proceedings of 10th International Workshop on Program Comprehension*, pages 114 – 123, 2002. 15

[SM95] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)* (Opio (Nice), France, October 16-20, 1995), 1995. 15, 22

[Sof] Software Architecture Toolkit. Website. http://www.swag.uwaterloo.ca/swagkit. 11, 36

[SSW02] Susan Elliott Sim, Margaret-Anne Storey, and Andreas Winter. A Structured Demonstration of Five Program Comprehension Tools: Lessons Learnt. In *Proceedings of 7th Working Conference on Reverse Engineering*, pages 210–212, 2002. 15

[ST81] M. N. S. Swamy and K. Thulasiraman. *Graphs, Networks, and Algorithms*. John Wiley & Sons, 1981. 17

[TG02] Qiang Tu and Michael W. Godfrey. An integrated Approach for Studying Architectural Evolution. In *Proceedings of 10th International Workshop on Program Comprehension*, pages 127–136, 2002. 10, 36

[TGLH02] John B. Tran, Michael W. Godfrey, Eric H.S. Lee, and Richard C. Holt. Architectural Repair of Open Source Software. In *Proc. of IWPC*, 2002. 10

[TH99] John B. Tran and R. C. Holt. Forward and reverse repair of software architecture. In *Proc. of CASCON 1999*, Toronto, November 1999. 11, 30

[TH00] Vassilios Tzerpos and R. C. Holt. ACDC: An Algorithm for Comrehension-Driven Clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 258–267, 2000. 12, 15

[WCR] WCRE Working Conference on Reverse Engineering. Website. http://reengineer.org/wcre2002/. 10

[WS00] Jingwei Wu and Margaret-Anne D. Storey. A Multi-Perspective Software Visualization Environment. In *Proceedings of CASCON*, 2000. 85